Mathematical Modelling Techniques in Process Design

Andrew T. Doig

Master of Philosophy
University of Edinburgh
1991

DECLARATION

The work described in this Thesis is the original work of the author and was carried out without the assistance of others, except where explicit credit is given in the text. It has not been submitted, in whole or in part, for any other degree at any University.

Andrew T. Doig

ABSTRACT

Software has been developed which constructs mathematical models and simulations of chemical engineering problems. It uses a generic description of each problem domain, *e.g.* a flash problem consists of mass and heat balances, vapour-liquid equilibrium relationships, *etc*, and a set of global constants, such as Antoine coefficients. The fixed variables must be supplied for each instance of the problem.

The first step in producing a simulation is the assignment of an equation to be solved for each variable in the problem; these may be design variables or others whose values are required. This assignment is found by the use of a flow maximisation technique. Next the equations are partitioned into their minimal solvable subsets by a depth first search algorithm. Following partitioning, the smallest set of variables such that, knowing their values, the rest can be calculated is identified, a guess is made for their starting values, and a computer program is written to solve the equations. This program uses the Newton Raphson method with analytical derivatives to solve simultaneous equation sets; the values of these derivatives are found without explicit differentiation using an extension of a method due to Ponton [75] for torn systems. Finally the results of computation are reported to the user.

Criterea are presented for the comparison of models and simulations, and qualitative definitions of merit are presented. The structural analysis of equation sets is discussed in detail, and common methods are described and contrasted. Throughout the thesis these topics are treated graph theoretically since many of the concepts considered are visualised most easily in this way. In particular one set of theorems appears which relate graphs, digraphs and their properties to the structure of equation sets, another shows how a flow maximisation technique can be used to solve the assignment problem, and yet another proves how and why the decomposition technique chosen works. Whilst the first two sets of results are well known, no proof has been located of them in the form in which they appear. No statement or proof for the last set of theorems has been found.

Finally some improvements to the software are proposed. These are concerned both with its structural detail, and with its ability to reason.

Acknowledgements

This work has been possible only due to the efforts of many people whose help I am delighted to be able to acknowledge. I am grateful to my supervisors Dr. Ken M^{c}Kinnon and Professor Jack Ponton, and my industrial sponsors, B.P.. So too immeasurable thanks are due to my parents, Andy and Rena Doig, and my brother, David, for their support during my studies, particularly during my period of illness. I am indebted to Carleen Robertson and Martin Fallon, for buying the beers and providing a calming influence. Last, but by no means least, I wish to thank my friends and colleagues in both the Department of Chemical Engineering and Pollock Halls for their help and advice.

Contents

List of Figures

Rules and models destroy genius and art.
William Hazlitt, On Taste

Chapter 1

The Requirements of a Modelling System

1.1 Introduction

In general, the computation of the answer to a numerical problem is a two stage process: firstly a mathematical model is formed, and then it is solved. Although there are many techniques available for the latter task, there is a dearth of theory which deals with the former. The production of a mathematical model can be troublesome, and both skill and experience may be necessary to construct one. Perhaps the first text to address itself to this impediment was Polya's [74] classic book, but this was a prescription for solving general mathematical problems rather than the production of models. This problem has been recognised on a wider scale [7] and it has prompted Aris [6] to publish a textbook on the fundamentals of mathematical modelling. Although this text addresses itself to a wider audience, it draws all of its substantial examples from the field of chemical engineering.

This thesis is more specific than Aris's text in that it is an investigation of some of the more important principles and practises of the mathematical modelling of chemical engineering problems, rather than models in general. The domain of application is even more restricted than this, because we will deal only with modelling single plant items and the physical and thermal changes which take place within them, not in the simulation of entire chemical plants; the problems associated with this larger scale modelling have been addressed by Hutton [47]. We will see how the formulation and interpretation of mathematical models can be decomposed into several areas - ranging from the selection of equations of the appropriate type to checking the results supplied by a computer program - and an account will appear of the problems associated with each of these tasks, and of the attempts made to address them. In § 1.2 the terms mathematical model and simulation are defined and contrasted, and a discussion of how examples of these may be compared appears in § 1.3.1. A modeller's toolkit is described in § 1.5. This must allow for the formulation of a model; its development to a simulation; the realisation of this simulation as a computational program; and a check and report of its results. Finally, § 1.5 indicates which of the problems in the preceding section have been addressed, and where their solutions appear.

1.2 Fundamental Aspects of Mathematical Modelling

This section details the elements of a good mathematical model, and the simulations which may be derived from it. Before proceeding with this discussion it is necessary to define these terms.

The journals of mathematics and the philosophy of science are littered with definitions of the term "model" [52], [5]. The most useful definition for our purposes is provided by Smith [87], who regards a model as a generic mathematical description of a problem. The adaptation of a model to describe a specific problem he terms a simulation. Thus one might model an exothermic reactor by writing down the differential and algebraic equations which describe it and simulate it by specifying what the reactants are to be, their inlet temperature and the fractional conversion of the key component, etc.. The solution to the problem is found by manipulating the simulation in such a way that the values of its dependent variables are determined. Throughout this thesis the term "model" is used in Smith's sense but the term "simulation" is extended slightly to cover the order in which information is to be used. Thus two models of the same problem differ if they use different sets of mathematical equations and different simulations of the same model can be produced by rearranging the information or the values of some of the constants used within it.

1.3 The Comparison of Models and Simulations

1.3.1 What is a Good Model ?

What is it that makes one formulation of a problem superior to another? The contention that the accuracy and superiority of models are synonymous is vitiated by considering a model of a reciprocating compressor. The most accurate

model of this system which can be imagined involves a description of how the molecules within the piston react with those on the cylinder walls and those within the entrained fluid. A large number of algebraic, differential and statistical expressions would be required to represent the system, and comprehension of such a model is unlikely to be easy. However, it is improbable that an engineer would require such a detailed description of the problem; it is far more likely that he would be interested only in the macroscopic properties of the system, and so he may well be content to model the compressor by using some relatively straightforward thermodynamic relationships and a simplified version of the Navier Stokes equations.

The important point to grasp is that the more accurate model contains too much information. The provision of this extra information is an inefficient use of the modeller's time, a barrier to a clear appreciation of the more salient aspects of the model, and an impediment to its solution. This is a pathological case, but it demonstrates the possibility of excessive rigour. This possibility exists, even for less extreme examples, when the data to be used are known to be inaccurate. If this is so then there may be little point in producing a finely detailed model since the results which it will yield will be of questionable value. The obverse of this is that a model which uses the ideal gas law may be insufficiently accurate for the engineer and thus one which uses, say, the Peng-Robinson equation [71] may be preferred. The obvious, but none the less vital, point to be stressed is that a 'good' model is one which uses only as much information as is necessary. Thus it is necessary to define one's level of interest before writing a mathematical model.

This argument demonstrates the difficulty of defining optimality in the context of mathematical modelling. It is tempting to define optimality in terms of the

amount of effort required to solve the problem - the faster the solution, the better the formulation, but this is unsuccessful. Not only does this definition fail to account for the appropriate accuracy of the model, but so too it neglects the amount of effort required to set the model up. It is impossible to provide a precise definition of optimality which encompasses all three of these points because it is difficult to define a meaningful estimate of the effort required to produce a mathematical model, and it is difficult to define a general measure of accuracy.

Despite these difficulties, some definition of optimality is required, albeit a fuzzy one, in order to allow at least a qualitative discussion of the relative merits of different models. Thus we will define a good formulation of a problem to be one which requires minimal overall effort to set up and solve whilst providing a suitably correct answer with sufficient clarity for the modeller to understand it.

It may be possible to discriminate between models which satisfy the above conditions. Consider, for example, a distillation column which is used to separate a feed of N components into S different streams. In order to avoid redundancy, any model which describes this system may contain at most $N + S + 1$ of the possible $N + S + 2$ mass balance equations. Suppose that during the formulation of the model $N + S$ of the mass balance equations have been used and that one of the remaining two is required to complete the description of the column. If, for instance, the two remaining mass balance equations were

$$\sum_{i=1}^{i=N} z_i = 1 \tag{1.1}$$

$$\sum_{j=1}^{j=S} W_j x_{j_k} = F z_k \tag{1.2}$$

where z_k is the mole fraction of the k^{th} component in the feed stream F, W_j

is the j^{th} product stream and x_{j_k} is the mole fraction of the k^{th} component in the j^{th} product stream[1], then either equation could be used without prejudice to the final overall knowledge contained within the model. This is so because any $N + S + 1$ of the mass balance equations may be used to derive the other.

Although the two models contain the same information, implicitly if not explicitly, they are different because of the equations used. It may be that neither model appears to be any better or worse than the other but important differences in their structure may come to light when the models are extended to become simulations. For instance, equation 1.1 would, in general, be easier to rearrange to give a new subject than would equation 1.2. Further, the first equation is linear whereas the second is likely to contain a number of bilinear[2] and, in most of its rearranged forms, non-linear terms; since, generally, linear equations are easier to solve than non-linear equations it may well be that a simulation which uses equation 1.1 is superior to that which uses equation 1.2. If the choice for the last mass balance equation had been between equation 1.1 and the mole fraction balance on the j^{th} stream

$$\sum_{i=1}^{i=N} x_{j_i} = 1 \tag{1.3}$$

then it would not be possible to select the better equation cannot by reference to equation form alone. However, the general heuristic is that one ought to use linear equations in preference to others wherever possible.

Lastly, one may compare simulations by the order in which they use information. For example, if the equations are to be solved by a Gauss-Seidel iteration, then the order in which variables are updated may determine the course of the solution.

[1] *n.b.* 'k' in equation 1.2 is used for generality. It would have to have been set to some particular value at this point.

[2] a bilinear term is a linear expression such as $\alpha * \beta$ where both α and β are variables

Further, if some of the variables are to be torn so that at each pass the values of some of the variables in the problem are determined by the solution of a 'kernel' problem, and the others found by direct substitution, different tearing strategies would produce different simulations. The solutions to these formulations would proceed in different ways and so they may exhibit distinct rates and stability of convergence.

It has been demonstrated that it is very difficult to provide precise, practical rules for discriminating between models and simulations, but that they may be contrasted according to inexact criteria. One can postulate the synthesis of an optimal simulation by manipulating these criteria in such a way that a score is ascribed to each of the above choices, a good choice being assigned a high score, and choosing the simulation which scores most highly. This is impractical, however, because even if a meaningful score could be given to each choice, the decision tree for even a small problem is likely to be very large. Hence, in practise, only a qualitative *a priori* comparison of models and simulations is possible.

1.4 The Derivation of a Mathematical Model

In the last section we discussed the nature of a good model. In this section we turn our attention to its production. This problem can be decomposed into four tasks, namely

1. Select the appropriate equation set.

2. Manipulate it into the desired form.

3. Develop a computational procedure for its solution[3].

4. Solve the problem, check the results and report them.

A further stage in the process which may be a practical necessity, or at least advisable, is the production of an approximation to the required model. We will deal with each of these tasks in turn.

1.4.1 Choosing the Appropriate Equation Set

The natural inclination of the engineer on encountering a problem is to make a diagram to represent it, and to jot down some of the variables associated with it. The next thing that he does is write down some of the the relationships which exist between these variables, e.g. heat and mass balances, thermodynamic relationships, fluid flow equations, etc. There may be little choice involved in the selection of some of these equations, e.g. the balance equations, but selecting the others may well involve skill and experience; for example, the choice of an equation of state and of physical property equations is a complicated enough task for expert systems to have been written to tackle it [8]. Having described the system in such general terms, the engineer must decide which of these equations are to be used in the model. Some expansion and contraction of the equation set will be necessary - for instance too many mass balance equations may have been

[3] *n.b.* in general this is not necessarily a computer program, but we restrict it to this definition for the purposes of our discussion.

provided and, possibly, some equations will be required whose necessity was not evident originally.

At this stage it is imperative that one be cognizant of the necessity for completeness and, as far as possible, consistency, and that one avoids the perils of redundancy. A complete set of equations is one in which there is a one to one correspondence between the equations and the variables which appear in them; the reasons for this condition are given in § 2.3, and methods for checking it are described in § 3.2. The term consistency refers to the assumptions which have been made about the system under consideration. In general, these should not conflict sharply if meaningful results are to be derived from the model, but this is not always the case.

Redundancy, which was touched on earlier when it was noted that an engineer may provide too many mass balance equations in a model, is a much harder problem with which to deal. Any equation set to be solved must be linearly and non-linearly independent, i.e. no equation may be expressed as either a linear or non-linear combination of some or all of the other equations. The reason for this is that if the value of any variable is to be determined it must be done by using some statement which has been made about the problem. For instance it may be that the temperature rise experienced by a fluid flowing through a heat exchanger can be calculated by using the equation

$$Q = UA\Delta T \tag{1.4}$$

If there are N such variables whose values are to be determined, then N such expressions must be provided. Suppose that during the compilation of an equation set E, ν equations have been used, and that a candidate for the next

equation to be included can be expressed as a combination of κ of the members of E. The inclusion of this equation provides no new information about the problem and so only ν of the $\nu+1$ members of V, the set of variables which corresponds to E, may be solved for. In this case the $(\nu+1)^{st}$ equation is said to be redundant and another equation must be selected in its place. Spotting that an equation set exhibits redundancy can be hard; determining the set of candidates for the redundant equation is extremely difficult.

1.4.2 Equation Manipulation

Once it has been established that a set of equations gives a complete, consistent and non-redundant description of a problem the next problem is to manipulate it into a simulation. Having chosen the values of the constants in the problem, there are four ways in which this can be done.

1. The form of the equations can be changed, e.g. logs can be taken of both sides of an equation which involves exponential terms (this is a standard trick in reaction equilibrium problems).

2. The equations can be rearranged into some form, *e.g.* $f(x) - b = 0$ or $x = f(x)$.

3. The equations may be reordered, and/or torn.

4. The equations can be differentiated analytically. This is necessary when a first or second order solution method is used since, in these cases, the Jacobian and/or the Hessian of the system is required.

For convenience, altering the form of an equation introduces at least one new variable and one new equation to the problem; e.g. taking the logs of both sides of the reaction equilibrium equation for a single reaction involving ideal gases

$$K_p = \prod_{i=1}^{i=N} P_i^{\nu_i} \tag{1.5}$$

produces the two equations

$$S = \sum_{i=1}^{i=N} \nu_i \log P_i \tag{1.6}$$

$$K_p = \exp S \tag{1.7}$$

This task involves a few simple rules, $\prod \rightarrow \sum$, $a^b \rightarrow b \log a$, etc, which can wielded relatively simply. Differentiating equations is an order of magnitude greater in difficulty, principally because there are many more rules involved; chain ruling is easy but flattening differentiated expressions can be intricate and troublesome. Harder yet is the rearrangement of equations to give them a new subject. It is easy to cope with finding an explicit expression for x from

$$y = \cos(\sqrt{x}) \tag{1.8}$$

but it is harder to derive one from

$$y = \frac{x}{1-x} \tag{1.9}$$

and impossible to manipulate

$$y = x + \log x \tag{1.10}$$

into the desired form. There are a few popular symbolic algebra packages available [77], [76], but although they can perform simple tasks very well, it is my experience with Macsyma that it is hard to use, easily confused and bad at recovering from a computational disaster.

Reordering equations is a simple task but, as we will see in § 3.2, finding the rearrangement which satisfies some criteria may require considerable effort. Firstly a decision must be taken as to whether the equations are to be partitioned into smaller subsets, and if they are to be decomposed or not. Some of the the theoretical basis required to answer these questions appears in § 2.5 and § 2.6. Secondly the equations may be solved by successive substitution, by a method which requires function values, e.g. the secant method, or one which uses derivatives too[4]. The best method to use is a function of the shape of the equations and the starting point for the solution, but even given knowledge of these data, it is difficult to discern the best strategy. When a choice of solution method has been made some questions remain; *e.g.*, if a derivative method is to be used how are the equations to be differentiated?, which decomposition strategy is best?, do we have to stay in the feasible region at all times? The first of these questions is discussed in § 6.3, and the second is considered both in § 2.6 and § 3.4.

1.4.3 Program Writing

Having decided on a solution strategy, *e.g.* that the problem is to be decomposed and that the kernel problem is to be solved using the Newton Raphson method, and having rearranged the equations as necessary, the next stage in the formulation of the simulation is to produce a computer program which will carry out the calculations. This is an algorithmic task. The main computational block must be written along with any subroutines that are necessary, the file compiled

[4]Many of these methods are described in appendix D

and linked with the system mathematics library and the whole program executed. This must be done with care. For example, two points which must be borne in mind are:

1. In the early stages of the formulation we may be dealing with vectors and matrices by referring to their members in general terms. For example, if we were modelling a single reaction taking place in an ideal vapour phase, we might choose to represent the N vapour phase mole fractions by

$$y_i = \frac{n_i^0 + \chi\nu_i}{\sum_{j=1}^{j=N}(n_j^0 + \chi\nu_j)} \tag{1.11}$$

   where n_i^0 is the number of moles of the i^{th} component originally present and χ is the extent of reaction. If the values of y_i are to be calculated simultaneously, a loop must be provided in the program; this requires that a new variable be invented for use as a count variable. This count variable must be distinct both from that which is used to perform the summation in the denominator of equation 1.11, and any other variables which control loops within which that for y_i is nested. Handling matrices requires a simple extension of the rules for handling vectors.

2. Attention must be paid to the idiosyncrasies of the language in which the program is to be written. If it is Fortran, then one must take care not to violate the restrictions concerning variable names which are inherent in that language; integers must be given names which start with a letter in the range I to N inclusive; variables of all other types must be given names which begin with a letter outwith this range. When one is writing in C one must recall that the language is case sensitive.

Generating a computer program automatically is a simple algorithmic task, but it may require a great deal of effort if the equations are complicated, and if a sophisticated language is used. This point is discussed in greater detail in § 6.7.

1.4.4 Finding, Checking and Reporting Results

It is unwise to believe that just because we have translated the mathematical description of a problem into a language like Fortran that compiling and running the resultant program will provide a correct answer; indeed it is naïve to assume even that it will provide an answer. It is important to be able to identify the mathematical causes of failure, should it occur, such as divergence from a solution or convergence to one which is physically infeasible where, for instance, an attempt may be made to find the logarithm of a negative number. Even when a program runs successfully the answer which it provides may be incorrect; there is no guarantee that an iteration will converge and, even if it does there is no guarantee that the solution will lie in the feasible region. Thus the numbers provided by a program must be checked to make sure that values lie within their logical bounds, e.g. temperatures are positive and mole fractions sum to one.

1.4.5 Approximation

In § 1.3.1 we touched on the need to provide a description of a problem at an appropriate level, the argument being that there is only so much of interest

within it. In this section we expand on the concept of finite description in order to establish the desirability and, in some cases, necessity of providing an approximate model of a problem.

The desirability of approximation is shown quite clearly by considering the mass balance over a plug flow reactor in which benzene is being hydrogenated to cyclohexane. The full mass balance equation is [21]

$$\frac{\partial x}{\partial t} + (\nabla^t u)x + \frac{R}{C_0} = D^t(\nabla^2 x) \tag{1.12}$$

where x is the fractional conversion of benzene, u is the gas velocity, R is the reaction rate, C_0 is the initial concentration of benzene and D is the effective diffusivity of the benzene. Solving this partial differential equation would be both difficult and expensive. If, however, we assume that the reactor is running at a steady state, and that the reactants and products are well mixed at each point along its length, the reactor can be modelled approximately by

$$(\nabla^t u)x + \frac{R}{C_0} = D_z \frac{\partial^2 x}{\partial z^2} \tag{1.13}$$

which is much easier to solve. If we go one step further and assume that the radial and angular variation of the velocity is small (remember that it is a packed bed) we can reduce the equation to

$$\frac{\partial u_z}{\partial z} x + \frac{R}{C_0} = D_z \frac{\partial^2 x}{\partial z^2} \tag{1.14}$$

which is even easier to solve. Equation 1.14 might be used either to replace 1.12 altogether or to give an initial solution to the problem which can then be used as the starting point to find the solution to the more complicated equation.

This may be an important technique when the equations to be solved are very non-linear. Consider, for instance, a model of an oil reservoir. Very detailed

vapour liquid equilibria calculations are required for this and so multi-termed equations of state must be used. Converging these from an arbitrary starting point may be extremely difficult, or even impossible. In this case the solution to a linear, or less non-linear, model may be necessary in order to provide a good starting point for the less tractable equation set.

1.5 A Modeller's Toolkit

Having discussed the definition of a good model and the tasks which are necessary to form it, we are able now to describe the tools which must be present in a mathematical modelling package. Since, in general, the equations used in a mathematical model are not unique, the software must be able to discriminate between alternatives in such a way that it produces a complete, non-redundant equation set which describes the system being modelled without any significantly conflicting assumptions. Having constructed such a set, it should be able to order it and to parse and rearrange its members into any desired form. So too it should have the ability to differentiate the equations, and identify a tear set from the variables if necessary. Lastly it should contain program writing and execution facilities, and an interpreter for checking and reporting the results.

A prototypical mathematical modelling system has been developed which includes some of the above features. No attempt has been made to provide an ability for qualitative reasoning, *i.e.* to compare formulations or to check solutions, and so only a single strategy is followed. This is described fully in § 6.7, but a brief summary of its components, and where they are dealt with in this thesis, is given

here.

Firstly a general set of equations is collated, along with a list of the fundamental and specific constants for the problem. These equations are stored in Reverse Polish Notation, and this is described in § 6.7. Next a complete subset of these equations is selected which will be used for the model, and a one to one correspondence between them and the variables in the problem is developed. An explanation of this process appears in § 4, and so too does a description of the way in which this equation set is partitioned into a sequence of smaller sets. In the next stage of the formulation, these sets are decomposed; this is described in § 5. Finally a C program is written which solves these equations by using a Newton Raphson method to accelerate the convergence of the tear variables. A novel data management technique is used to find the numerical values of the analytical derivatives of the tear equations, and this is described in § 6.

In § 2 the rudiments of graph theory are described and an attempt is made to define some conditions on an equation set for it to have a unique solution. So too in that chapter the need for the selection of an output set is explained. Further, a graph theoretical description of matrix partitioning appears. This is used to show that the minimal, solvable subsets of an equation set are independent of the output set selected. Lastly, the tearing of equation sets is discussed. In particular some definitions of optimality are examined and the effect of tearing on the efficiency of some numerical techniques is considered.

Some of the practical techniques which have been used to examine the structural phenomena described in § 2 are reviewed in § 3. First techniques for choosing an output set are considered and then our attention is focused on partitioning

matrices in order to find their lower triangular form. This work is extended to irreducible matrices in order to characterise fill-in in symmetric and asymmetric matrices. Finally decomposition strategies and algorithms are discussed. One method of each form of analysis - output set selection, partitioning and tearing - was selected for use in the modelling software. Their choices are vindicated in this chapter.

In § 4 we justify the decision to find an output set for a problem and then to partition the equations. Next the reduction of a general, possibly inconsistent model to a consistent and specific form is considered. In this section an observation is made about how redundancy in equation sets can be overcome. This is followed by a proof that a maximum flow technique can be used to find the output set and a statement and discussion of Dinic's maximal flow algorithm [20]. The final point dealt with in this chapter is the ordering of the equations so that they form minimal, solvable subsets. The depth first search algorithm of Tarjan [94] is presented and analysed.

The search for a minimum cardinality tear set is considered in § 5. First it is shown that a minimal cardinality tear set for a signal flowgraph is also one for the bipartite digraph from which it was derived. Next we show that a search for that tear set can be reduced to the roots of a spanning forest of the signal flowgraph. Finally algorithms for forming a signal flowgraph from a bipartite digraph, finding its spanning forest and then searching for a minimum cardinality tear set are presented and discussed.

Analytical differentiation is examined in § 6. An extension to torn systems of Ponton's method for generating the numerical value of analytical derivatives

is described and illustrated in an example. In § 6.7 the construction and functionality of the modelling software is described. Details are given of the data structures used and the way in which an abstract statement of a problem is transformed first into an expanded algebraic form, and how this is then used to generate a solution. An example which was solved by the software is provided. Lastly a summary of the conclusions drawn from the separate chapters is presented in § 8.

We will restrict ourselves to modelling steady state systems which are described by sets of algebraic equations. This condition precludes the necessity of examining the specific solution requirements of differential and integral equations. Further, the models produced will be for solution on a serial computer.

Angling may be said to be like the mathematics, that it can never be fully learnt

Izaak Walton, The Compleat Angler

Chapter 2

The Graphical Analysis of the Structure of Equation Sets

2.1 Introduction

In the last chapter it was stated without proof that it was desirable to describe a simulation problem using a square equation set, *i.e.* one in which there are exactly as many equations as there are variables within them. Further, it was asserted that an output set should be chosen for the equation set, that the equations should be permuted into a sequence of smaller subsets where possible and that any of these sets which contain two or more equations should be decomposed. In this chapter we turn our attention to the justification of these assertions and examine some of the ways in which the desired goals may be achieved.

Considerable use is made of graph theory in this and other chapters and so we begin with a summary of the graphical definitions, and the properties of graphs

which are germane to our discussion. The desirability of solving square sets of equations is addressed in § 2.3 and the need to find a one to one correspondence between the variables and the equations in a problem is discussed in § 2.4. The way in which the partitioning of matrices and the permutation of their rows and columns relate to solving equation sets is described in § 2.5. Here vertex elimination from a graph is used as an analogue for the effect of ordering equations, and some comments appear about the effect on the rate of convergence to a solution of a set of equations of different permutations. § 2.6 contains a definition of decomposition and a discussion about the nature of a 'good' tear set. An analysis of the effect of decomposition on the amount of effort required at each iteration for a range of solution methods appears in § 2.6.2. This range is not comprehensive, but it is large enough to show that there is a considerable number of numerical methods for which there is no 'structural' advantage in tearing. A summary of the conclusions drawn from § 2 appears in § 2.7.

Prior to discussing graph theory, it is necessary to relate this subject to equation solving. Consider the following equation set.

$$\begin{aligned} x_1 &+ 2x_2 - x_3 = 1 \\ 2x_1 &- x_2 \quad\quad = 2 \\ &\quad x_2 + x_3 = 3 \end{aligned} \tag{2.1}$$

The graph of equations 2.1 is shown in figure 2.1. Anticipating the terminology of § 2.2.1, each variable and equation in 2.1 contributes a node to figure 2.1. The line drawn between nodes x_1 and F_1 indicates that variable x_1 appears in equation F_1. Representing the equation set in this way permits one to reason about its structure so that, for instance, one can determine the dependency of one variable upon another.

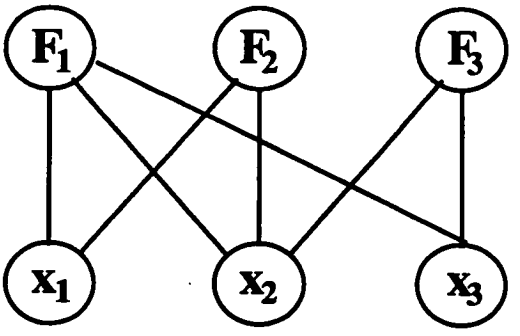

Figure 2.1: A Graph of the Equations

2.2 Graph Theory

2.2.1 The Elements of a Graph

Consider two objects, u and v, and a relationship, e, which is defined between them. If these objects and their relationship are represented pictorially as in figure 2.2 then u and v are termed *nodes* and e is called an *arc*. These nodes

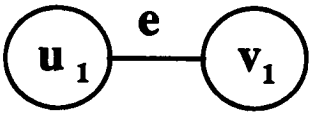

Figure 2.2: A Graph

may also be called *vertices* and the arc may be called an *edge*; these alternatives will be used interchangeably throughout the chapter. If the same relationship which exists between u and v can be defined between other nodes as well then the set of all nodes is called V, the set of all edges is called E and the structure

which includes them all is the *graph* $\mathcal{G}(V, E)$. This definition may be extended to include a single node, which is the minimal non-null graph. Since there is an edge between u and v in figure 2.2, they are said to be adjacent to one another and e may be written as the unordered pair (u, v). Later we will return to consider the case where (u, v) is an ordered pair. If this edge is traversed from u to v then e is said to be *incident* from u to v.

A *path* in $\mathcal{G}(V, E)$ is a set of vertices $p = \{v_1, v_2, \cdots, v_k\}$ such that $v_i \in V$ and $(v_i, v_{i+1}) \in E, i = 1, 2, \cdots, k-1$. If $v_1 = v_k$ then p is said to be a *cycle* or, equivalently, a *circuit.* A path, p, is said to be *simple* if no vertex, ν_j, or edge, e, appears in it more than once. Similarly, if the initial vertex, ν_1, of a cycle, c, is the only vertex to appear in it more than once, if this node appears exactly twice, and if no edge appears in c more than once, then it is a simple cycle.

If $V' \subseteq V$, $E' \subseteq E$, and $u, v \in V' \forall (u, v) \in E'$, then the graph $\mathcal{G}'(V', E')$ is a *subgraph* of $\mathcal{G}(V, E)$. Two vertices v_i and v_j are said to be *connected* if there is an undirected path from v_i to v_j ; further each vertex is connected to itself. Any subgraph $\mathcal{G}'(V', E')$ of $\mathcal{G}(V, E)$ in which each $v_i \in V'$ is connected to each $v_j \in V'$, no $v_k \in V'$ is connected to any $v_m \notin V'$, and such that $\forall v_i, v_j \in V' and (v_i, v_j) \in E$, $(v_i, v_j) \in E'$ is called a *component* of $\mathcal{G}(V, E)$. In the chemical engineering literature this is referred to as a *partition* of the graph. Clearly connection is an equivalence relation on vertices, and $\mathcal{G}(V, E)$ may be partitioned into a set of subgraphs

$$G \rightarrow \{\mathcal{G}_1, \mathcal{G}_2, \cdots, \mathcal{G}_\mu\} \tag{2.2}$$

such that the vertices and edges of each $\mathcal{G}_i$ are distinct. If each vertex $\nu_i \in V$ in a graph $\mathcal{G}(V, E)$ is adjacent to every other vertex $\nu_j \in V$, then $\mathcal{G}(V, E)$ is the *complete* graph on V. The complete graph on some subset $\overline{V} \subseteq V$, *i.e.*

$\overline{G}(\overline{V},\overline{E})$, $\overline{E} = \{(u,v) \mid u,v \in \overline{V}\}$, is said to be a *clique.* In figure 2.3, $\{\nu_1,\nu_2,\nu_3\}$ is a simple path and $\{\nu_4,\nu_5,\nu_6,\nu_7,\nu_4\}$ is a simple circuit. This graph has one component, namely itself, and vertices ν_1, ν_2, and ν_3 form a clique.

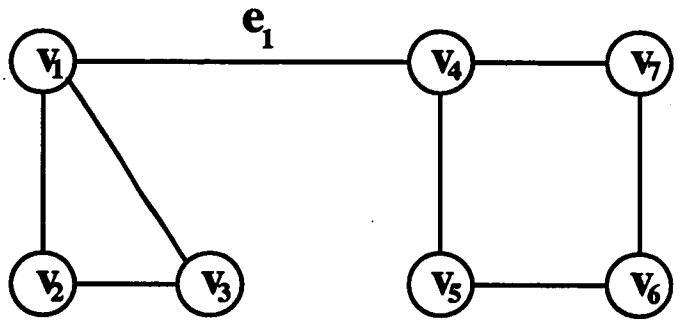

Figure 2.3: A Single Component Graph

If the removal of a node $v_k \in V$ from some graph $\mathcal{G}(V,E)$ breaks any of the circuits in $\mathcal{G}$ then v_i is said to be an *articulation point* or a *separator* of $\mathcal{G}$. A set of articulation points, S, such that each cycle in $\mathcal{G}(V,E)$ has at least one node in it, is a separation of $\mathcal{G}$; in chemical engineering texts this is referred to as a *tear set.* If instead of nodes, edges are removed from $\mathcal{G}$ and the maximum number of these is removed which allows all $v_i, v_j \in V$ which are connected in $\mathcal{G}(V,E)$ to remain connected in $\mathcal{G}(V,E'), E' \subseteq E$, then this latter graph is said to be a *minimum spanning subgraph* of $\mathcal{G}$. The minimum spanning subgraph for figure 2.3 is $\mathcal{G}(V,E')$, where $E' = E - \{e_1\}$. Any connected graph which contains no circuits is called a *tree.* The vertex which is ordered first in this tree is called its root, and each connected subgraph which is formed by deleting the root and the edges incident from it, such that there is at least one edge in the subgraph, is referred to as a *branch* of the tree. Each of these connected subgraphs is itself a tree and, so it too can be said to have a root and, possibly, branches. Any

connected subgraph of a tree in which there are no edges, *i.e.* a single vertex, is a *leaf* of the tree. A *forest* is a graph in which each component is a tree. It follows from these definitions that the minimum spanning subgraph of a connected graph is a tree, whereas that for an unconnected graph is a forest.

We are now in a position to define those classes of graph which are of interest to us. Prior to this, however, it is worthwhile relating those properties of graphs just described to the equations and variables which they represent. In a graph which corresponds to an equation set there is an edge between nodes v_i and u_j if variable v_i appears in equation u_j. For example, returning to our consideration of figure 2.1, the edge between nodes x_3 and F_1 indicates that variable x_3 appears in the first equation. If two nodes which correspond to variables appear in a simple circuit, then the equation used to solve for either requires the value of the other, and so these equations must be solved simultaneously or an algebraic substitution made of one variable for the other. Developing this argument shows that all of the nodes in a circuit which represent equations must be solved simultaneously and so a component of a graph represents a subset of the equations which must be solved together. A proof of this appears in § 2.3. We proceed now to classify the types of graph with which we will deal.

2.2.2 The Types of Graph of Interest

Since the equation sets will always be finite, so too will be the graphs used to represent them. In most cases the edges in $\mathcal{G}(V, E)$ will have a particular direction associated with them, *i.e.* (u, v) will be an ordered pair. Such graphs are termed

directed graphs, or *digraphs*, and the *strong component* is the equivalence class. This is defined analogously to the component of an undirected graph in that there is a directed path from each node ν_i to each other node in the same strong component, and a directed path from each of these back to ν_i; no such pair of paths exist for nodes which belong to different components. The number of strong components of a digraph $\mathcal{D}(V, \acute{E})$ may be greater than the number of components of the underlying undirected graph as figure 2.4 shows, but the converse can never be true. Here, letting C_i be the set of nodes in the i^{th} strong component,

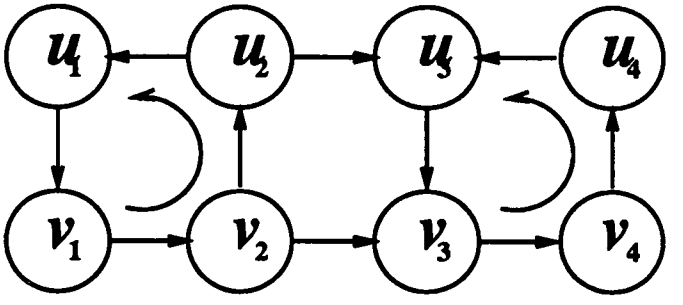

Figure 2.4: A Digraph which has two Strong Components

$C_1 = \{v_1, v_2, u_1, u_2\}$ and $C_2 = \{v_3, v_4, u_3, u_4\}$ whereas the underlying undirected graph of figure 2.4 has only one component. Analogously to the definitions given in§ 2.2.1, the minimum spanning subdigraph of a connected directed graph is a *directed tree* and that for an unconnected digraph is a *directed forest.*

In general there will be no *parallel* edges, *i.e.* multiple edges between two nodes which are oriented in the same direction. Further, except for one class of graphs, no node will direct an edge onto itself, *i.e.* there will be no *self loops.* Any graph which features neither parallel edges nor self loops is called a *simple graph.*

If the vertices of $\mathcal{G}(V, E)$, whether $\mathcal{G}$ is directed or not, can be partitioned into m

distinct sets, V_i, such that there are no edges between any two vertices $v_i, v_j \in V_l$, then $\mathcal{G}(V, E)$ is said to be an *m-partite graph.* The most important example of a graph of this type is the ***bipartite graph***, *i.e.* $m = 2$. Lastly, if $\mathcal{D}(V, \acute{E})$ is a bipartite digraph such that $V = V_x \cup V_y$, $V_x \cap V_y = 0$, then the digraph $\mathcal{H}(V, \tilde{E})$,

$$\tilde{E} = \{(u, v) \mid u, v \in V_x, (u, w), (w, v) \in E\} \tag{2.3}$$

is called a *signal flowgraph* . This digraph can be thought of as a 'condensation' of $\mathcal{D}(V, \acute{E})$ in that its strong components correspond to those of $\mathcal{G}$ but that the nodes of V_y are excluded; $\mathcal{H}(V_y, E')$, the signal flowgraph which results from excluding the nodes of V_x, is defined similarly.

As an example of the graphs discussed above, consider the equation set which is used to model an ideal, binary flash problem in appendix B. The undirected graph of these equations is shown in figure 2.5, where each numbered node represents an equation. This graph shows which variables appear in each equation. Figure 2.6

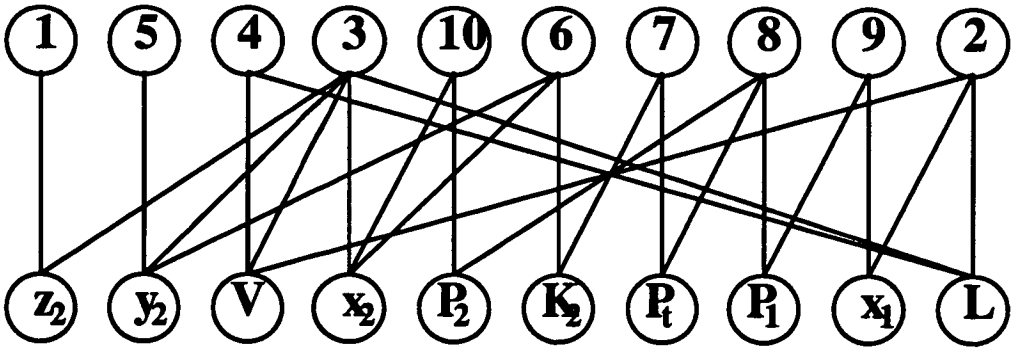

Figure 2.5: The Graph of the Flash Equations

is a directed version of this graph; as will be shown in § 3 this corresponds to choosing to rearrange each equation so that it is solved for one of the variables within it. In this digraph, an edge is directed from an equation node, ν, onto a

variable node, ω, if the corresponding equation is to be solved for the variable represented by ω. Note that both of these graphs are bipartite. Finally, figure 2.7

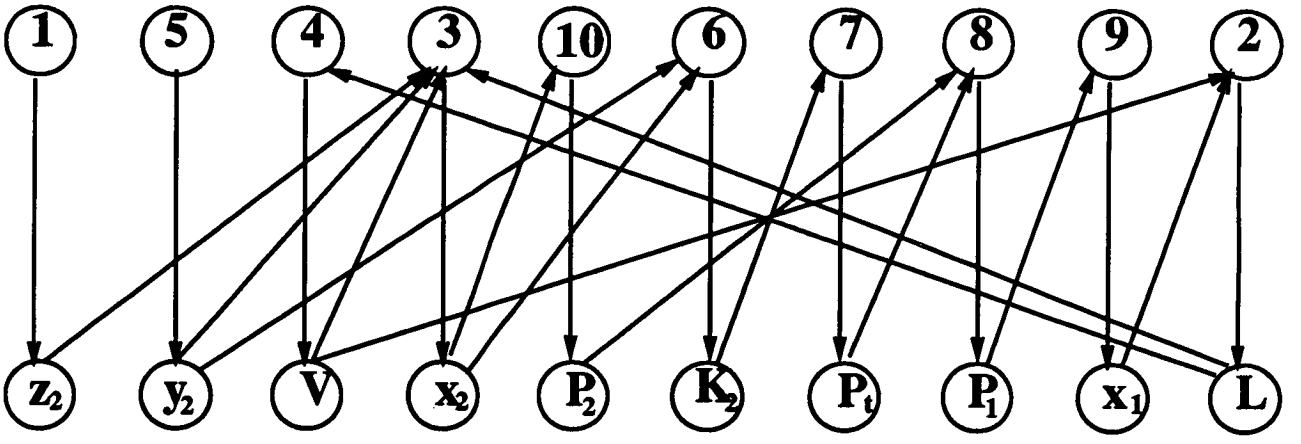

Figure 2.6: A Digraph for an Assignment of the Flash Equations

is the signal flowgraph which corresponds to figure 2.6. This signal flowgraph

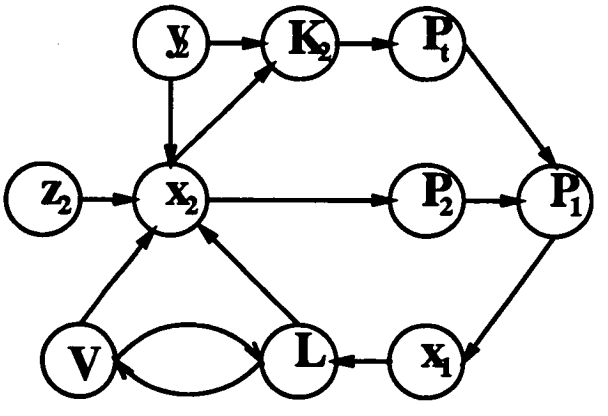

Figure 2.7: The Signal Flowgraph for the Flash Equations

demonstrates that, *e.g.*, y_2 and x_2 appear explicitly in the equation to be solved for K_2. As will be shown in §5, each circuit in a signal flowgraph, $\mathcal{H}(V, \tilde{E})$, corresponds to one in $\mathcal{D}(V, \acute{E})$, the bipartite digraph from which it is obtained.

2.2.3 The Properties of Graphs

The *degree* of a vertex, $d(v)$, is the number of edges to which it is connected. If $\mathcal{D}(V, \acute{E})$ is a digraph then

$$d(v_i) = d_+(v_i) + d_-(v_i) \tag{2.4}$$

where $d_+(v_i)$ is the *in-degree* of v_i, *i.e.* the number of edges directed to it, and $d_-(v_i)$ is the *out-degree* of v_i, the number of edges directed away from this node. For instance, the *in*-degree of node 2 in figure 2.6 is two, whereas its *out*-degree is one.

An *ordering* of the nodes in a graph is the assignment of an ordinal number in the range $1 \rightarrow N$ to each of the N nodes in the graph. We will define a partition of the graph to be an ordering such that the ordinals for the nodes in a component are contiguous and, for $\mathcal{D}(V, \acute{E})$ directed, for any pair of vertices v_i and v_j which belong to different strong components and for which $i < j$ there is no directed path from v_j to v_i. This corresponds to an ordering of the strong components of $\mathcal{D}(V, \acute{E})$ such that, as we will see in theorem 2.1, the equation subsets which they represent can be solved sequentially.

Consider a subset of edges $M \subseteq E$ in the graph $\mathcal{G}(V, E)$. If the endpoints of the members of M are pairwise disjoint, *i.e.* no vertex is the endpoint of more than one edge, then M is said to be a *matching* in $\mathcal{G}$. The largest such subset possible is called a *maximal cardinality* matching in $\mathcal{G}$ and if each $v_i \in V$ is an endpoint of one of the edges in M, then it is said to be *complete*. If the edges of $\mathcal{G}$ have weights assigned to them then a matching with the largest possible sum of weights from E is called a *maximum weight matching*. It is important to note

that a maximal matching in a bipartite graph is a one to one correspondence between the items represented by the smaller vertex subset and a subset of items represented by the larger set, but this does not imply that each vertex in the smaller set appears in the matching. This is a point to which we will return in § 3.2.

2.2.4 Vertex Elimination

The process of removing some node $\nu \in V$ from an unipartite graph or digraph, $\mathcal{G}(V, E)$, and adding sufficient edges that each path of length $l \geq 2$ which passed through ν in $\mathcal{G}(V, E)$ becomes a path of length $l-1$ in the new (di)graph is called *vertex elimination.* This process short circuits each path in $\mathcal{G}(V, E)$ in that each path $\{\omega, \nu, \sigma\}$ is replaced by $\{\omega, \sigma\}$. If the edges (ω, ν) and (ν, σ) were present in $\mathcal{G}(V, E)$, then the edge (ω, σ) is present in the new graph; if this edge was not in the original graph, then it is said to have filled in. Vertex elimination on bipartite digraphs must be defined differently in order to avoid violating the condition that no vertex may be adjacent to another in the same partition. In this case elimination must be considered on pairs of matched vertices and paths of length $l \geq 3$ which pass through them are replaced by paths of length $l-2$ in the new digraph. This phenomenon is demonstrated by considering the bipartite digraph in figure 2.8. If nodes e_1 and v_1 are eliminated, then the edge (v_4, e_2) fills in. Should this graph be replaced by its corresponding signal flowgraph, then vertex elimination in this graph, which is defined as for other unipartite graphs, would provide a corresponding fill edge between nodes 4 and 2, regardless of the set of vertices on which the signal flowgraph is based; this phenomenon is

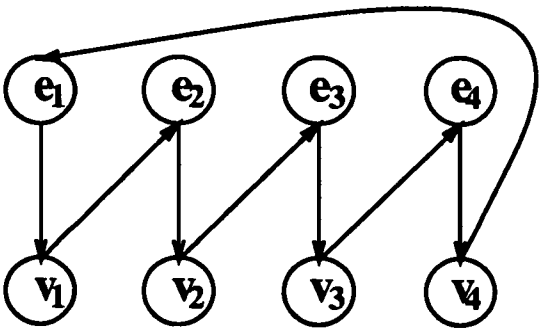

Figure 2.8: A Bipartite Directed Graph

demonstrated in figure 2.9(a), where the broken arc signifies the filled edge. This

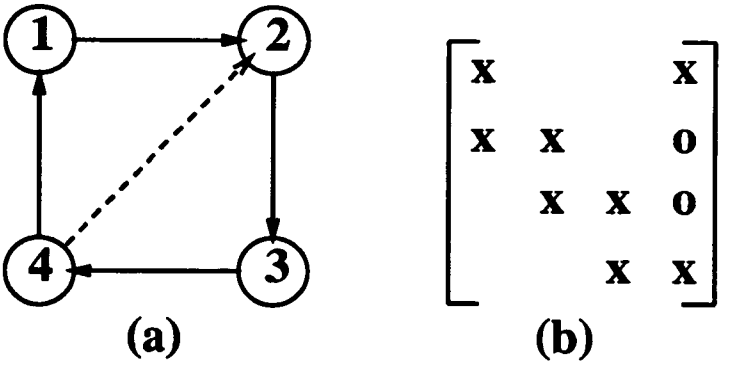

Figure 2.9: Vertex Elimination on a Digraph

phenomenon can be used to describe the way in which information is chained through an equation set.

If an equation set is represented by a digraph, $\mathcal{D}(V, \acute{E})$, then each of the variables which is represented by a node in a cycle of $\mathcal{D}(V, \acute{E})$ is dependent on each of the others. In general, the first variable is an explicit function of some of those ordered later and the second variable is a similar function of later variables and, possibly, the first. This pattern is repeated for each variable represented in the

cycle. Consider some variable x_j which is an explicit function of some other variables. If one of those variables, x_i say, is ordered before x_j, then x_j is an implicit function of the variables in which x_i is explicit. If eliminating x_i from the signal flowgraph which represents the equation set leads to the addition of any edges, then these signify the implicit dependency of some of the variables in the problem. Consider, for example, figure 2.9(a), which is the signal flowgraph that is derived from the bipartite digraph of figure 2.8. Removing node 1 produces a fill edge $(4,2)$, so that the second variable is implicitly dependent on the fourth. This is demonstrated most clearly by considering Gaussian Elimination. Here, the fill-in pattern produced within the matrix corresponds exactly to the filled edges of the graph which represents it. As an example of this, figure 2.9(b) is the matrix which corresponds to figure 2.9(a). Here **x** represents a non-zero in the original matrix, and + represents a filled entry. This is explained more fully in §2.5.

2.2.5 Graph Representation and Algorithmic Complexity

In order to relate graph theory to the computer solution of equation sets, a brief description of the matrix representation of a graph and a discussion of computational complexity are necessary. An adjacency matrix of a graph is a matrix in which each column, k, which has a 1 in the i^{th} row, corresponds to a node, k, for which the edge, (k,i), exists in $\mathcal{G}(V,E)$. If $\mathcal{G}(V,E)$ is undirected this matrix is symmetric but if $\mathcal{G}$ is directed, then each row j which has a non-zero entry in column, l, represents an edge, (l,j), in $\mathcal{G}$. Note that in the bipartite

case, even when $\mathcal{G}$ is undirected, if the rows and columns for the nodes in each partition are ordered consecutively, each of the entries in both the upper left and lower right quadrants is zero. If the other two quadrants are superimposed so that each row in the new matrix represents an equation, and so that each column corresponds to a variable, then the result is referred to as an *incidence matrix.* This arises from the fact that each non-zero entry in a row represents a variable in the corresponding equation. As an example, figure 2.10 is the incidence matrix for the graph in figure 2.5. Finally, each column, k, of the adjacency matrix may be

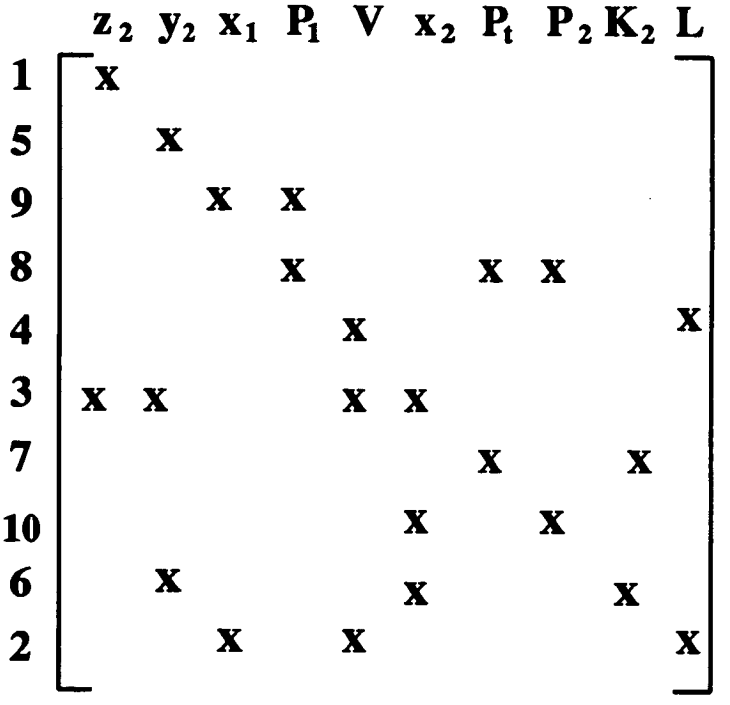

Figure 2.10: The Incidence Matrix for the Flash Equations

rerepresented by $Adj(v_k)$, the *adjacency set* for node, k. This is the set of vertices which lie at the endpoints of the arcs which emanate from v_k. For example, in figure 2.5, $Adj(x_1) = \{1, 8\}$ and the column in the matrix of figure 2.10 which corresponds to x_1 has non-zero entries only in the rows labelled one and eight.

The *complexity* of an algorithm is a measure of the number of operations required to execute it, and hence of its efficiency. This is expressed by its *order*, a function which relates the time taken for its execution to the size of the problem being solved. If for some algorithm this function is ψ, then the order of the algorithm is written as $O(\psi)$; ψ can be a constant, a polynomial, a factorial or a transcendental function. This is an inexact measurement because it assumes that all operations take the same time and only its worst case value is calculated. Despite this, used with a knowledge of its shortcomings, it is an invaluable tool in the analysis of computational algorithms. The following description is restricted to analysing graphical algorithms, but the definitions and concepts provided are applicable to the entire domain of computation.

In general, an algorithm is regarded as efficient if its time complexity can be expressed as a low order polynomial; *e.g.* Tarjan's depth first search algorithm [94], which is described in § 3.3.3, is $O(N + \tau)$, where there are N nodes in the graph and τ arcs, and Dijkstra's shortest path algorithm [19] is $O((\tau + N)log_eN)$.

The class of decision problems which can be solved by polynomial time algorithms is called P. There is another class of decision problems for which no deterministic polynomial time algorithm has been found, but for which the verification of a solution lies in P; this class is known as NP.Consider some decision problem Π_1. A *polynomial transformation* from Π_1 is a function $\mathcal{F}$ which translates any instance of Π_1 into an instance of another problem Π_2 such that the answer to Π_2 is 'yes' if and only if the answer to Π_1 is yes, and such that $\mathcal{F}$ can be computed efficiently. Any problem Π_i which belongs to the subset of NP such that there is a polynomial transformation from Π_i into each other problem Π_j in the subset,

and from Π_j into Π_i, is said to *NP-complete.* This is a large and important set of problems and if a polynomial time algorithm is found for one of these, then, by definition, a polynomial time algorithm will have been found for them all. This point is raised again in § 3.3.1.

§ 2.2 has described the most basic and general components of graph theory. Some more definitions and concepts are required but they are introduced later as required.

2.3 Conditions for a Unique Solution

In this section an attempt is made to define those conditions on an equation set which are necessary or sufficient for it to have an unique solution. Four properties of the equations are examined:

1. The number of equations to be solved and the number of variables within them.
2. The structure of the equation set, *i.e.* the interdependence of variables and equations.
3. The algebraic structure of the equations.
4. The degree of nonlinearity of the functions over the domain of the solution.

Consider the solution of M equations in N variables. If M is less than N then the system is underdetermined in that there are $N - M$ too few constraints on the values which the variables can take. There is a set of problems which are underdetermined and which can be solved uniquely, but each of these solutions is trivial. For example, $x^2 + y^2 = 0$ has a unique solution at $x = y = 0$. In general, however, there may be an infinite number of solutions to an underdetermined equation set. If, on the other hand, M is greater than N then there are $M - N$ too many constraints on the values which the variables can take. There is no guarantee that these superfluous constraints can be satisfied at the same point as the N others; such a system is said to be over determined and it may have none, one or many solutions. Should M equal N then a unique solution may exist because under these circumstances it is possible to provide a one to one correspondence between the variables and the items of information provided by the equations. Hence there is no condition on the relative sizes of the equation set and the set of variables within it, which is either necessary or sufficient for a unique solution of the equations to exist.

Pantelides [68] has indicated that one consequence of Hall's [37] theorem of combinatorics is that a necessary condition for a unique, non-trivial solution of a system of N equations in N variables by successive substitution, is that every subset of k of the N equations must contain at least k variables. Should this condition be violated then the system is said to be structurally singular . For instance, since equations 2.5 are three equations in only two variables, they violate this condition. Even if this condition holds there will still be no unique solution to the problem if one or more of the equations is redundant, *i.e.* it can be expressed as a combination of l of the others. Once more equations 2.5 provide an example of this; the first and last equations may be multiplied to give

the second. Thus another necessary condition on the uniqueness of solution is the requirement that the equation set should be non-redundant. Neither of the above conditions is sufficient for a unique, non-trivial solution, however, because structurally non-singular and algebraically non-redundant equation sets may be numerically singular over part of their domain. This occurs when two or more equation surfaces become parallel over some region in space. Thus uniqueness of solution requires that such regions be avoided.

$$\begin{aligned} x_1^2 - 2x_2^2 &= 0 \\ 2x_1^3 - x_2x_1^2 - 4x_1x_2^2 + 2x_2^3 &= 0 \\ 2x_1 - x_2 &= 0 \end{aligned} \tag{2.5}$$

Two necessary conditions for an equation set to have a unique solution have been established, but no useful sufficient conditions have been found for the solution of general, non-linear equation sets. We will return to this problem in § 4.2.

2.4 The Need to Select an Output Set

Let E be a set of equations in the variables in the set X and, further, let $|E| = |X|$. Let the set of ordered pairs P,

$$P = \{(e_i, x_i) \mid e_i \in E, x_i \in X, i = 1, 2, \cdots, |E|\} \tag{2.6}$$

be a legal one to one correspondence between E and X, *i.e.* that the i^{th} equation, e_i, contains at least one occurrence of the i^{th} variable, x_i.[1] Then P is said to

[1] **n.b.** The subscript i refers to an ordering of each equation and variable in P, not in E or X.

be an output set for the problem. Each of the pairs in P represents an equation which can be solved for a given variable. One assignment for the flash problem of appendix B is

$$P = \{(1, z_2), (2, L), (3, x_2), (4, V), (5, y_2), (6, K_2), (7, P_t), (8, P_1), (9, x_1), (10, P_2)\} \tag{2.7}$$

As has been indicated above, such an assignment is possible if and only if the problem is not structurally singular, and thus the determination of an output set may be used as a check on this condition.

However there are two other reasons for selecting an output set. Firstly, if the set of equations is to be solved by successive substitution then each equation must be rearranged to an explicit form for a given variable; choosing an output set ensures that this is done legally. Although in its simplest form it is a poor solution method, this strategy can be developed to others which are of some merit, as is shown on page 75. Secondly, if a matrix method, *e.g.* Newton Raphson, is to be used, the selection of an output set must be carried out so that the adjacency matrix of the graph of this problem can be permuted to have a zero free diagonal; in this case the output set is called a maximum transversal. As is shown below, the failure to permute this matrix to this form may cause some partitioning algorithms to fail.

In general an output set for a given problem is not unique. Lemma 2.1 provides an upper bound on the number of possible output sets for a set of equations E.

Lemma 2.1 *If E is not structurally singular then, $\mathcal{S}$, the number of possible*

output sets is bounded by

$$0 \leq \mathcal{S} \leq \mid E \mid ! \tag{2.8}$$

Proof: The proof of the lower bound is trivial. Let $N = \mid E \mid$. By definition, in order for an output set to exist, there must be an equivalent number of equations and variables in the system. Clearly, the greatest number of output sets possible occurs when each variable appears in each equation. It is sufficient to show that there are $N!$ possible output sets when this condition holds. In this case there are N choices for the variable to be solved for by the first equation, $N-1$ choices for the second, $N-2$ for the third, and so on until only one choice remains for the last equation. Regardless of the choice of variables for the first k equations the remaining $N-k$ equations can be assigned to the remaining $N-k$ variables in each of the $(N-k)$! possible independent ways. Thus the upper bound on the number of output sets for an equation set E is $\mid E \mid !$. □

As will be described in the next chapter, some of these output sets may be preferable to others, but there is no known means by which a set which is known *a priori* to be optimal in any given sense, may be selected.

2.5 The Nature of Partitioning Matrices

In this section we will discuss the permutation of incidence matrices. The definition of graph partitioning given towards the end of § 2.2 extends naturally to the incidence matrices in that the rows and columns of these matrices correspond

to the nodes in the graph; reordering the nodes simply reorders the rows and columns. In essence there are many forms to which a matrix may be permuted but, as we will see in § 3.3, there is only one way to partition a given matrix given the strict definition of this term. Of all of the possible forms, only four are of interest here:

1. Banded matrices. An example of one of the more common banded matrices, the tridiagonal matrix, is shown in figure 2.11(a). This form is of particular use in the solution of the algebraic equations which arise from the discretisation of partial differential equations.

2. Lower (Upper) triangular matrices as shown in figure 2.11(b). Permuting a matrix to this form allows the exact, non-iterative solution of the equation set in the forward (backward) direction.

3. Block diagonal form. As shown in figure 2.11(c) all of the blocks which straddle the diagonal are square and no non-zero entries appear above these blocks. This is a weaker form of (b).

4. Bordered Diagonal form. This is a weakening of the structure of (c) as is clear from figure 2.11 (d). This form is used frequently, especially when the diagonal blocks are of unit size, *i.e.* when the matrix is of bordered lower diagonal form.

The tridiagonal form will not be dealt with further but the properties of matrices of types (b), (c) and (d) will be dealt with after the relationship between the strong components of a graph and the structure of the corresponding equation set has been established.

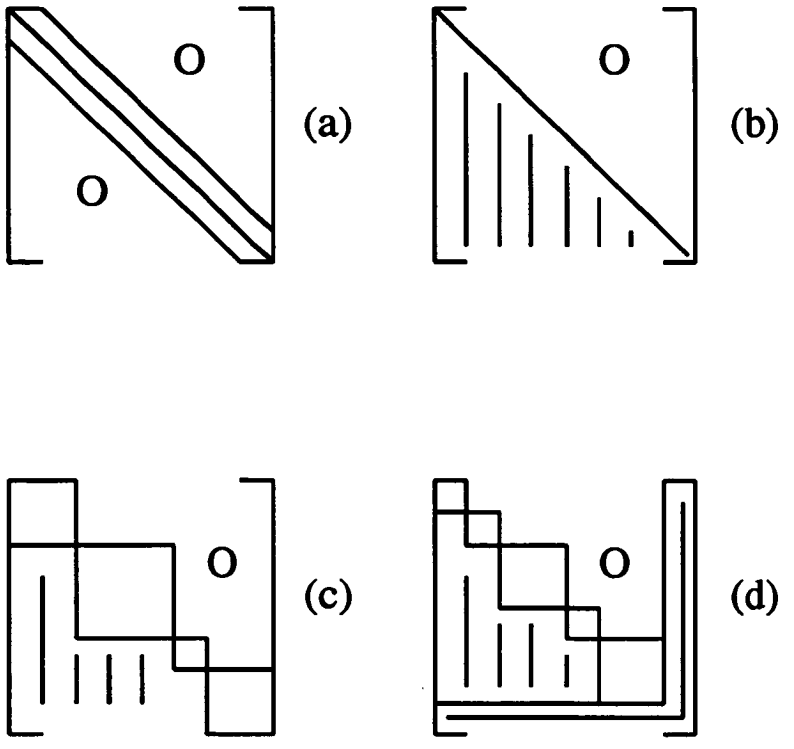

Figure 2.11: Four Desirable Matrix Forms

Consider the 4 × 4 equation set 2.9. The digraph for one transversal of this equation set is shown in figure 2.12.

$$\begin{aligned} x_1 + x_2 &= 3 \\ x_1^2 - log_e(\tfrac{x_1}{x_2}) + x_3 + x_4 &= 4 \\ exp^{-(x_1^2 - \sqrt{x_2})} + x_3 - x_4 &= 5 \\ x_1 - x_2 &= 1 \end{aligned} \tag{2.9}$$

The nodes in this digraph which correspond to variables are labelled with the name of the variable, and those which correspond to equations are labelled E_i, according to the order in 2.9. The strong components of this digraph are $C_1 = \{x_1, x_2, E_1, E_4\}$ and $C_2 = \{x_3, x_4, E_2, E_3\}$. The incidence matrix for this digraph appears in figure 2.13(a).

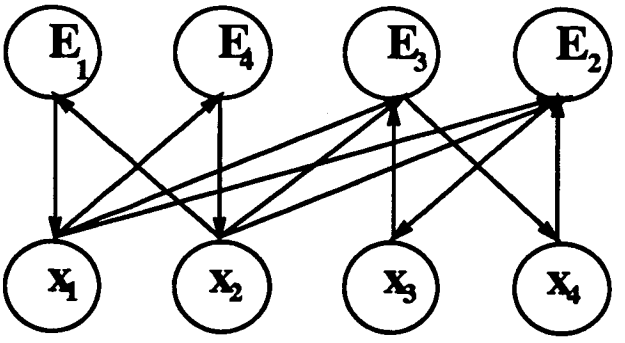

Figure 2.12: The Digraph of the 4×4 Equation Set

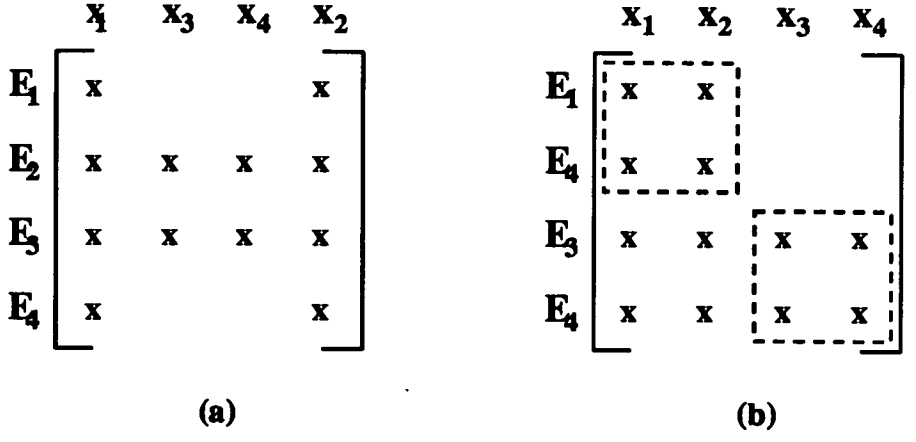

Figure 2.13: The Incidence Matrix for the 4×4 Equation Set

This matrix can be partitioned into the form of figure 2.13(b), which shows that equations E_1 and E_4 can be solved simultaneously for x_1 and x_2 before the remaining two equations are solved simultaneously for x_3 and x_4, using the exact values of x_1 and x_2. This grouping of variables and equations is called a computational sequence for the equation set. This term is defined as an ordering of equation subsets such that each is a solvable system of equations of minimal size, and such that they may be solved sequentially. Thus no equation may be removed from a member of a computational sequence and leave a solvable

subset, and no equation may be dependent on a variable which is solved for in a subset ordered later in the sequence. As an example, $\{C_1, C_2\}$ is a computational sequence for equations 2.9, but $\{C_2, C_1\}$ is not, and neither is $\{C_1, C_3, C_4\}$, where $C_3 \cup C_4 = C_2$.

We wish to show that a computational sequence for an equation set is unique to within some well defined, allowable permutations. To do this we will demonstrate that it corresponds to an ordering of the strong components of the digraph which represents the assigned equation set, in which there are no edges from a strong component to another which is ordered before it. By inspection, it can be seen that, for equations 2.9, $\{C_1, C_2\}$ satisfies these conditions, whereas neither $\{C_2, C_1\}$ nor $\{C_1, C_3, C_4\}$ does. The general case is explained by the following remarks, observations, lemmas and theorems.

We begin with some general observations about the relationship between equation sets, graphs and digraphs. As has been noted already, an equation set $F(X)$ can be represented by a bipartite, undirected graph $\mathcal{G}(V, E)$, where V is the union of $\hat{V}$, which corresponds to the equations, and $\check{V}$, which represents the variables within them; each edge $(\hat{\nu}, \check{\omega}) \in E$ denotes that $\check{\omega}$ is one of the unknowns in $\hat{\nu}$. This graph contains no information about which equation is to be solved for which variable. If a complete matching, $M \subseteq E$, exists for the graph then this can be used to form a directed bipartite graph $\mathcal{D}(V, \acute{E})$ such that there is a one to one correspondence between A and E; each $(\hat{\nu}, \check{\omega}) \in M$ becomes the directed edge $(\hat{\nu}, \check{\omega})$ in $\mathcal{D}$ and every other edge in E is directed in the opposite direction in the new graph. The interpretation placed upon a directed edge $(\hat{\nu}, \check{\omega}) \in A$ is that equation ν is to be solved for variable ω.

We aim to show that the strong components of $\mathcal{D}(V, \acute{E})$ correspond to the minimal equation subsets into which $F(X)$ may be partitioned, and that these strong components may be ordered in such a way that they define a computational sequence. As a first step we show that the strong components of this digraph may be ordered so that there is at least one which has no edges directed onto it from another, and at least one from which no edges are directed. Next we show that in each strong component, the numbers of nodes from each vertex set are equal and that any subset of k nodes from $\hat{V}$ directs exactly k edges onto nodes form $\check{V}$. These results are used to show the correspondence between the strong components of $\mathcal{D}(V, \acute{E})$ and a computational sequence for $F(X)$. Lastly we prove that this computational sequence is independent of the complete matching used to form the directed graph.

Lemma 2.2 *It is always possible to order the strong components of a digraph $\mathcal{D}(V, \acute{E})$ so that if there is a path from some vertex ν in the i^{th} strong component, C_i, to a vertex ω in some other strong component C_j, then $i < j$, and there is always at least one strong component in the digraph which has no incoming edges, and one from which no edge is directed onto a vertex in another strong component.*

Proof: Consider $R(X, A)$, a reduction of $\mathcal{D}(V, \acute{E})$ in which the i^{th} strong component of the directed graph is represented by a node $\nu_i \in X$ and the arc set $A = \{(\nu_i, \nu_j)\}$ such that there is at least one edge in $\mathcal{D}(V, \acute{E})$ between a node in C_i and one in C_j. Any path through the vertices of $R(X, A)$ corresponds to a path in $\mathcal{D}(V, \acute{E})$ which passes through at least two strong components and so, by definition, $R(X, A)$ must be acyclic. Since $\mathcal{D}(V, \acute{E})$, and hence $R(X, A)$, is finite,

it follows that there must be at least one node in the reduced digraph from which no edge is directed. Let such a node be ν_k. If C_k is ordered as the last strong component of $\mathcal{D}(V, \acute{E})$ then any edge in this digraph which has as one of its termini a node in C_k must be directed from a lower to a higher ordered strong component. If ν_k and all of the edges incident upon it are deleted from $R(X, A)$ then there must be at least one node in the new digraph from which no edges are directed. Ordering the strong component of $\mathcal{D}(V, \acute{E})$ to which it corresponds second last retains the forward condition on the arcs of this digraph. This process of ordering and deleting can be continued until only one node remains in the reduced digraph. This node must represent the first strong component of $\mathcal{D}(V, \acute{E})$. No edges are incident upon this node and so there may be no edges incident upon the first strong component of $\mathcal{D}(V, \acute{E})$. □

This result will allow us to show that at least one subset of equations from $F(X)$ is independent of the others, and so it can be solved before them. The next lemma is required in order to show that a strong component of $\mathcal{D}(V, \acute{E})$ corresponds to a square, solvable subset of equations from $F(X)$.

Lemma 2.3 *If $\mathcal{D}(V, \acute{E})$ is a bipartite digraph which represents a square, structurally non-singular equation set, $F(X)$, such that $\hat{V}$ is the set of vertices which correspond to equations and $\check{V}$ is the set of vertices which represent variables, then the number of nodes in each strong component which are members of $\hat{V}$ is equivalent to the number of nodes in this subgraph which are members of $\check{V}$. Further each subset of k vertices from this strong component which are members of $\hat{V}$ direct exactly k edges onto the nodes in $\check{V}$.*

Proof: By definition it is possible to trace a circuit through each of the vertices in a strong component of the bipartite digraph $\mathcal{D}(V, \acute{E})$. This digraph is constructed in such a way that each $\nu \in \hat{V}$ has exactly one edge directed from it to some vertex $\omega \in \check{V}$, and hence any cycle which passes through ν must be extended through ω. Since, by construction, each $\omega \in \check{V}$ must be an endpoint of exactly one edge directed from some $\nu \in \hat{V}$, these vertices must appear an equal number of times in any cycle. Hence there must be exactly as many vertices from $\hat{V}$ in any strong component of $\mathcal{D}(V, \acute{E})$ as there are $\check{V}$. Further, since there is exactly one edge directed from each $\nu \in \hat{V}$, each subset of k nodes from $\hat{V}$ must direct k edges onto $\check{V}$. □

Lemma 2.4 *Each strong component of the digraph of lemma 2.3 represents a structurally non-singular, solvable subset of equations.*

Proof: This proof requires three observations.

1. Each node in a strong component is a member of a circuit in $\mathcal{D}(V, \acute{E})$ which involves all of the other vertices in that strong component.

2. There is no circuit in $\mathcal{D}(V, \acute{E})$ which involves two nodes ν_x and ν_y which lie in different strong components.

3. By lemma 2.3 in each strong component of $\mathcal{D}(V, \acute{E})$ there is an equal number of nodes from each partition of V, and each subset of k nodes from $\hat{V}$ directs exactly k edges onto vertices in $\check{V}$.

The first of these observations demonstrates the mutual dependency of the variables represented in a strong component, and hence the necessity for the corresponding equations to be solved simultaneously. The second shows that there can be no interdependence between two nodes ν_x and ν_y, which represent variables x and y respectively, where these lie in different strong components. This means that it is unnecessary to solve any other equation simultaneously with those represented in a strong component. Thus these equations, and only these equations, must be solved simultaneously. The last observation shows that their solution is possible since there are exactly as many equations in the system as there are unknowns, and each of these may be solved for one of the unknowns. □

Having established the preliminary results we can proceed to provide the formal correspondence between the strong components of $\mathcal{D}(V, \acute{E})$ and a computational sequence for $F(X)$.

Theorem 2.1 *Let $F(X)$ be an equation set such that $|F| = |X|$ and $F_i(X_i) \subseteq F(X), i = 1, 2, \cdots, M$ be a computational sequence for $F(X)$. Then, if $\mathcal{D}(V, \acute{E})$ is the bipartite digraph which represents $F(X)$, the M strong components of $\mathcal{D}(V, \acute{E})$ correspond to the subsets $F_i(X_i)$. Further, if these strong components are ordered so that each arc between two of them is directed from that which is ordered lower to that which is ordered higher, then ordering the equation subsets in the same way gives a computational sequence for $F(X)$.*

Proof: It is necessary and sufficient to demonstrate the following two properties of $\mathcal{D}(V, \acute{E})$. Firstly, the first strong component corresponds to a square,

structurally non-singular subset of $F(X)$. Secondly, all other strong components represent structurally non-singular equation subsets in which the number of variables is greater than or equal to the number of equations. Where this inequality holds, the nodes representing the excess variables belong to strong components numbered earlier, and so each strong component represents a square, structurally non-singular reduced subset of $F(X)$.

First Part: By lemma 2.4 the first strong component of $\mathcal{D}(V, \acute{E})$ represents a non-singular subset of equations and, from lemma 2.2, there are exactly as many variables in this set as there are equations.

Second Part: Once more, lemma 2.4 shows that each strong component of $\mathcal{D}(V, \acute{E})$ represents a non-singular subset of equations. In this case, however, there may be more variables than equations in the set. If there are K equations in the i^{th} subset then, by lemma 2.3, there are K nodes in the strong component which represent variables in these equations. All of the other variables in the subset are represented by edges from nodes in other strong components. Lemma 2.2 shows that all of these strong components can be ordered before the i^{th} one. Thus the value of each of these variables is known when the i^{th} subset is to be solved, and so this represents a square, non-singular, reduced subset of $F(X)$. □

This leads us to the following surprising result.

Theorem 2.2 *The computational scheme for an equation set $F(X)$ is independent of the output set.*

Proof: Let the undirected bipartite graph $\mathcal{G}(V,E)$ represent $F(X)$, and let M be a complete matching defined on its vertices. We wish to show that the strong components of $\mathcal{D}(M)$, the bipartite directed graph formed from $\mathcal{G}(V,E)$ and M in the manner of page 43, are independent of M, and that their ~~order~~ ordering which corresponds to a computational sequence for $F(X)$ is unique to within some trivial permutations.

First Part: Recall that M is a complete matching. Another complete matching for $\mathcal{G}(V,E)$, $\tilde{M}$ may be generated by removing some edge $(\hat{\nu},\check{\omega})$ from M, adding a new edge $(\hat{\nu},\check{u})$, removing $(\hat{\sigma},\check{u})$ and so on. Eventually some edge $(\hat{\alpha},\check{\omega})$ must be added to the new matching in order to complete it. If this process is repeated it can be used to generate all possible matchings for $\mathcal{G}(V,E)$.

Let the bipartite digraph formed from $\mathcal{G}(V,E)$ and $\tilde{M}$ be $\mathcal{D}(\tilde{M})$. Since there is a one to one correspondence between the edges of $\mathcal{D}(\tilde{M})$ and those of $\mathcal{G}(V,E)$, and between the edges of $\mathcal{D}(M)$ and those of $\mathcal{G}(V,E)$, this correspondence exists between the edges of the two digraphs. Any edge which is a member of both of these matchings, and any which is a member of neither, is directed in the same way in $\mathcal{D}(M)$ as it is in $\mathcal{D}(\tilde{M})$; any edge which is a member of only one of these matchings is oriented in opposite directions in the two digraphs. By construction, those edges in only one of the matchings must alternate in a cycle in each of the digraphs. Clearly this cycle cannot be extended through some edge (ν_x,ν_y) such that ν_x and ν_y belong to different strong components since there can be no edge directed back to the strong component of which ν_x is a member. Thus since each modification to M must describe a cycle through $\mathcal{D}(M)$, the strong components of each bipartite digraph formed from a matching in $\mathcal{G}(V,E)$ and the original graph must be the same.

Second Part: As has been shown above, the strong components for each complete matching in $\mathcal{G}(V, E)$, and hence for each output set for $F(X)$, are the same, and, by theorem 2.1 they may be ordered so that they correspond to a computational sequence for the equation set. Further, the edges between strong components in each of the bipartite digraphs must be oriented in the same direction because they represent the existence of a variable in an equation which can never be solved for it. Hence any ordering of the strong components of one of the bipartite digraphs which corresponds to a computational sequence is a similar ordering for each of the other digraphs.

If an ordering of the strong components of a bipartite digraph is to correspond to a computational sequence for $F(X)$, then it is necessary for each edge between vertices in different strong components to be directed from the lower to the higher ordered strong component. However, if there is no directed path between two strong components C_i and C_j in $\mathcal{D}(M)$, then either of these may be ordered before the other in any computational ordering. Hence, there may be more than one ordering of the strong components which satisfies the condition on directed paths between them, and hence more than one computational sequence for an equation set. □

We can extend the results from this proof to show that if an incidence matrix for an equation set is partitioned in such a way that the new matrix is block lower triangular and the diagonal blocks are irreducible, then the ordering of these blocks, and the rows and columns within them, is independent of the output set selected for $F(X)$. To do this, we establish that these diagonal blocks correspond to the strong components of the bipartite digraph which represents $F(X)$, and that they must be ordered in the same way as the strong components for the

new matrix to be block lower triangular. Finally, in theorem 2.5 we relate the uniqueness of the strong components of the digraph to that of the diagonal blocks. Here, a block triangularization of a matrix is a permutation of its rows and columns so that there are square, irreducible blocks on the diagonal, no non-zero entries above these blocks, and either zero or non-zero entries below them.

Theorem 2.3 *If $\mathcal{D}(V, \acute{E})$ is the bipartite digraph which represents an equation set $F(X)$, and if $A_{\mathcal{G}}$ is the incidence matrix of this digraph, then the strong components of $\mathcal{D}(V, \acute{E})$ correspond to the square diagonal blocks of a block triangularisation of $A_{\mathcal{G}}$.*

Proof: Let the rows of $A_{\mathcal{G}}$ represent the equations in $F(X)$, and the columns the variables. Order the rows and columns of $A_{\mathcal{G}}$ so that those which represent nodes in the same strong component are contiguous and so that those which appear in the first strong component appear before the second, and so on. Lemma 2.3 shows that, in each strong component, there is an equal number of variable and equation nodes. Each of these blocks must have a non-zero entry in its upper right entry, and so too it must be irreducible because it reflects the cycle structure of one of the strong components. No other permutation of the rows and columns within the matrix can produce different irreducible blocks, and so each strong component corresponds to a square block in the incidence matrix.

To see that these blocks are aligned along the diagonal of the matrix, consider that which corresponds to the first strong component. As is shown in lemma 2.2, the first strong component has no edge incident upon it from another. Hence, the block which represents it may have no rows above it nor columns to the

left of it, and so it lies on the diagonal. The block corresponding to the second strong component has its rows immediately below those for the first, and its columns immediately to its right; since this block too is square, it must straddle the diagonal of A_Q. Extending this analysis to the other strong components completes the proof. □

Theorem 2.4 *If A_Q is the incidence matrix associated with a square, structurally non-singular equation set, $F(X)$, then if it is partitioned so that there are square, irreducible blocks along its diagonal, these blocks correspond to a computational sequence for $F(X)$.*

Proof: Theorem 2.3 shows that these blocks correspond to the strong components of $\mathcal{D}(V, \dot{E})$, the digraph of $F(X)$, and theorem 2.1 shows that these strong components represent a computational sequence for this equation set. □

Theorem 2.5 *The rows and columns within the blocks of a block triangularization of a structurally non-singular matrix are independent of the permutations used to form them.*

Proof: Permuting the rows and columns within a structurally non-singular matrix in such a manner that it retains a zero free diagonal corresponds to reordering the vertices in the bipartite digraph which represents it, and, if the permutations are asymmetric, reorienting some of its edges. Theorem 2.3 shows that the diagonal blocks of a block triangularization of a matrix correspond to the strong components of this digraph, and theorem 2.2 indicates that the nodes within these strong components are independent of the ordering of the

vertices, or the orientation of the edges. Thus the rows and columns in any block triangularization of a structurally non-singular matrix are independent of the permutations used to form them. □

This result was anticipated by Steward [92] and proved in a different way by Duff [22].

The computational blocks may be reordered to some desirable form as will be described in §3.3. As they stand, however, these subblocks can be solved sequentially, and this ~~this~~ may effect both the amount of effort and storage which are required at each iteration, and the convergence characteristics of the equation set. If the equations to be solved are linear, or if they are linearised forms of non-linear equations, and if some matrix method is to be used to solve them, then permuting rows and columns may save fill-in. Minimisation of fill-in in $\mathcal{D}(V, \acute{E})$ during vertex elimination is important regardless of whether the equations being solved are linear or non-linear. If the equations are linear, then the explicit effect of fill-in is that entries are added to the factor matrices which were zero in the original matrix; this leads to an increase in the amount of storage and computation required for a solution. There is a similar effect if the equations are non-linear, although in this case the effect is implicit. Adding new arcs to the graph corresponds to chaining the values of some variables in the set through some equations in which, originally, they do not appear explicitly. Minimising fill-in minimises this coupling effect and so ought to lead to a more efficient solution. Should some gradient numerical method be used, then this chaining filters through to the derivatives of the equations; this is true both for linear and non-linear equations. As is shown in § 2.6, this has implications for the amount of work necessary when the equations are to be torn.

Partitioning the equations may have an effect on the convergence characteristics of the solution. It is likely that the equations in each of the subsets apart from the first will converge more quickly than it would when solved in the unpartitioned set. This arises from the fact that, for all connected subsets other than the first, partitioning the equations allows some of the erstwhile variables to appear as constants, their values having been calculated earlier. This effect is most noticeable if the equations to be solved are highly non-linear, since partitioning will increase their linearity. Consider, for instance, the solution of equations 2.9. The presence of the two transcendental functions causes this equation set to appear to be highly non-linear. Should the equation set be partitioned and the first and last equations solved simultaneously before the second and third, however, then the equation set is translated into the linear reduced system

$$\begin{aligned} x_1 + x_2 &= 3 \\ x_1 - x_2 &= 1 \\ C_1 + x_3 + x_4 &= 4 \\ C_2 + x_3 - x_4 &= 5 \end{aligned} \tag{2.10}$$

where $C_1 = x_1^2 - log_e(\frac{x_1}{x_2})$ and $C_2 = exp^{-(x_1^2 - \sqrt{x_2})}$ are constants. Equations 2.10 can be solved exactly, whereas equations 2.9 cannot. Further, removing the non-linear terms from the equation set ameliorates the use of any derivative information used in an iterative scheme, because it removes the possibility of divergence due to the variation in curvature of the equations over the domain of the solution. In general, partitioning an equation set will not be as successful in reducing the non-linearity of the equations to be solved as it was for the above example, but it is reasonable to expect some improvement. Whilst this may be of little advantage far away from the solution, its benefit is likely to increase as the search approaches it. This argument can be extended to the derivatives of the equations.

2.6 The Use of Decomposition Techniques

In the last section the partitioning of digraphs was discussed. Decomposition can be seen as an extension of this method which alters the strong components of a digraph. The aim of decomposition is the removal of nodes and arcs from some digraph $\mathcal{D}(V, \acute{E})$ in such a way that the modified digraph contains no circuits. This practice is known both as tearing and cutting, although the latter has a slightly less general meaning than the former. In this text all three terms will be used interchangeably and, although the following definition is given in terms of node tearing, a similar definition exists for edge cutting.

Formally, a decomposition strategy seeks to identify a node separator set, $S \subset V$, in a digraph, $\mathcal{D}(V, \acute{E})$, such that every cycle, C_j, in $\mathcal{D}(V, \acute{E})$ has at least one node in S such that for

$$\overline{E} = \{(u, v) \mid u, v \in (V - S), (u, v) \in E\} \tag{2.11}$$

$\mathcal{D}(V - S, \overline{E})$, is acyclic. For figure 2.6, for instance, $S = \{L\}$, and each edge which is directed to or from L is removed from E to give $\overline{E}$. Different orderings of the nodes in S and $V - S$ give rise to different orderings of the rows and columns of the incidence matrix of $\mathcal{D}(V, \acute{E})$. If the first node to be torn is placed at the end of the order, the next placed in the penultimate position and so on, then the incidence matrix thus produced has bordered lower triangular form. There is no unique tear set for a digraph and some may be larger than others. As we will see in § 3.4 these sets can be grouped into equivalence classes.

It is not clear whether it is preferable to tear an equation set before or after partitioning. Leigh [55] has shown that the number of tears for an unpartitioned

digraph is bounded below by the maximum of the minimum size of each of its strong components. Intuitively it is preferable to partition and then tear, since this exploits the natural structure of the equation set by grouping together those equations which are most strongly coupled. However, Sargent [82] provides an example in which fewer tears are required if the set is torn before rather after partitioning.

The most obvious benefit of tearing an equation set is that it reduces the number of variables whose values have to be guessed before the equations can be solved. The second advantage is that it can reduce the amount of computational effort required at each iteration during solution; this is a point to which we will return in § 2.6.2. It should not be assumed, however, that tearing an equation set is always worthwhile, since there is only a small class of numerical methods whose performance can be thus improved. Even when methods which lie within this class are employed tearing may be undesirable because of the effect which it has on the topology of the equations being solved. In order to provide a justification for the use of decomposition methods this section is divided into two parts. The first of these is a discussion of the nature of tear sets and, in particular, an attempt is made to define a 'good' tear set. Secondly we turn our attention to classifying those methods whose performance may be improved by decomposing the original equation set.

2.6.1 Optimal Tear Sets

As Motard *et al.* [64] have indicated there are no known optimum criteria for choosing tear sets. The most widely used strategies are those which minimise the number of torn nodes or edges, the weight of the torn arcs or, in flowsheeting problems, the number of recycle parameters, although this is simply a special case of the minimum weight approach. The weight of an arc is a value assigned to it, possibly in an arbitrary manner. One traditional method for assigning weights in flowsheeting problems has been to set them equal to the number of parameters associated with the corresponding process stream. Another assignment philosophy is described in § 3.2. Minimising the cardinality of the separator, S, is intuitively attractive because at each iteration it seems likely to lead to a more exact solution of the problem and a lower effort requirement than any larger tear set. Finding the tear set of lowest weight is an attempt to take into account the relative desirability of tearing each of the arcs in a digraph. In a flowsheeting environment this will generally correspond to a minimal amount of recycle information but in the wider field it may reflect the relative ease of solution of the equations; clearly, minimising the size of the separator is a special case of minimising its weight.

Even if an optimal tear set were to be of minimum size, minimising the size of this set, however this was defined, would be only a necessary condition for optimality. Upadhye and Grens [98] have suggested that the optimal tear set for a graph is likely to be nonredundant, *i.e.* no cycle in the digraph is torn by more than one edge or node in the tear set. Their argument is based on a consideration of the lag of information flow through the system being modelled by the graph. Their argument can be extended to say that, where possible, each cycle should be torn

the minimum number of times. Consider the two graphs shown in figure 2.14 where a bar, |, on an arc indicates that it is torn.

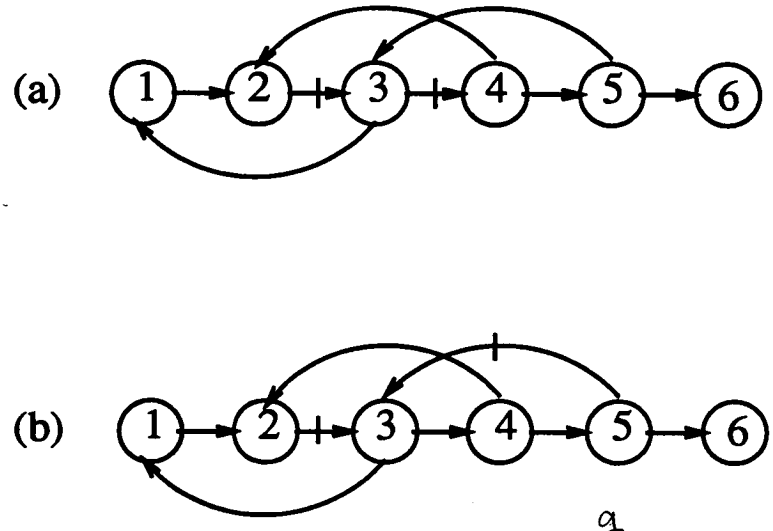

Figure 2.14: Two Tear Sets for a Digraph

Two minimum tear sets for these graphs are shown. In the first graph the cycle which involves nodes 2, 3, and 4 is torn twice. Here the value for second tear is updated without using the information which is available from the object represented by node three. In the second graph no cycle is torn more than once, and the information from node three is used. This ought to give the second iteration a superior rate of convergence because of the less arbitrary variation in the torn values.

As shown above, minimisation of the size of the tear set is insufficient for structural optimality. This strategy is also insufficient from an algebraic point of view because it takes no account of the effect of tearing on either the untorn equations, or those which are used to improve the guesses for the tear variables.

If possible, the tear set should be chosen so that it minimises the work done overall, *i.e.* it minimises the product of the number of iterations and the amount of work done at each, and so that it avoids singularities and discontinuities in the torn problem. Determining such a tear set is impossible at present because there is no sufficiently sophisticated method of algebraic analysis which allows this to be done efficiently.

2.6.2 Numerical Techniques Improved by Tearing

In this section the effect of tearing on the amount of effort required at each iteration for the following classes of numerical methods is considered:

- Direct Substitution.
- Relaxation Methods.
- Aitken's Method.
- Newton Like Methods.

Each class of numerical method is described fully in appendix 4 and so only a minimal description is provided here. In each case it will be assumed that $|x| = N$, that $c < N$ of the elements of x are torn and that each of the first $N - c$ equations has been rearranged to give an explicit expression for one of the $N - c$ dependent variables. The term 'full problem' will be used to mean the untorn form of the equations and, where appropriate, all subtraction operations will be

counted as additions, and all divisions as multiplications. The variable r_i, defined by

$$r_i = x_i - f_i \tag{2.12}$$

is used to denote the residual of each equation at the i^{th} iteration.

Direct Substitution Here the equations are written in a form which uses the value of the vector x at the i^{th} iteration to produce those at the $(i+1)^{th}$ iteration, *i.e.*

$$x^{i+1} = f(x^i) \tag{2.13}$$

If such an equation set is torn the calculations required at each iteration are

1. Evaluate the values of the dependent variables.
2. Evaluate the values of the independent variables.

Inspection of points 1 and 2 reveals that the steps involved in solving the torn equations are identical to those involved in solving the full problem. Thus there is no saving in computational expense or storage requirement associated with tearing an equation set if the equations are to be solved in this way.

Relaxation Methods The general form for calculating x^{i+1} with a relaxation method is

$$x^{i+1} = x^i - \omega^i r^i \tag{2.14}$$

where r^i, the residuals of the equations at the i^{th} iteration, are zero at the solution and ω^i is some acceleration factor; *n.b.* $\omega = 1$ corresponds to the method of direct

substitution. Three methods of calculating ω^i are described.

Successive Over Relaxation (SOR) The equations are solved in two stages. First of all the residuals are calculated and then x^i is updated by

$$x^{i+1} = x^i - \omega\, r^i \tag{2.15}$$

where $\omega \geq 1$. In the torn case, only the torn variables are updated. Since ω is a constant factor, only $N - c$ multiplication/subtraction pairs are saved per iteration. This is unlikely to be a significant saving in effort compared with either the cost of the function evaluations or the cost of determining the tear set.

The Secant Method This method accelerates the direct substitution method described in equation 2.13. It uses a different acceleration factor for each member of x, *i.e.*

$$x^\star = x^i + \omega_j\ (x_j^{i+1} - x_j^i) \tag{2.16}$$

$$\omega_j = \frac{1}{1 - s_j}, j = 1, 2, \cdots, N \tag{2.17}$$

$$s_j = \frac{f_j(x^{i+1}) - f_j(x^i)}{x_j^{i+1} - x_j^i}, j = 1, 2, \cdots, N \tag{2.18}$$

Tearing an equation set in this case reduces the computational expense of acceleration from $3N$ multiplications and $5N$ additions to $3c$ multiplications and $5c$ additions. This saving may be significant, particularly if $c \ll N$ and the equations are linear.

The Dominant Eigenvalue Method (DEM) This method is similar to the secant method in that it accelerates the solution to equations 2.13. These are

solved successively until the largest change in the elements of x occurs at an approximately constant rate. When this occurs an acceleration step

$$x^\star = x^i + \frac{x^{i+1} - x^i}{1 - M} \tag{2.19}$$

is taken, where M is the ratio of the largest change in an element of x at successive iterations. At each non-accelerating iteration, tearing the equations saves $N - c$ subtractions in finding the largest change in a variable over the course of the iteration. On acceleration, $N - c$ divisions and 2 $(N - c)$ additions are saved. Neither of these reductions in effort is likely to be significant.

Aitken's Method Aitken's method operates directly on the variables, and it ignores their interaction. Once again the direct substitution equations are solved but this time, when the difference in the value of a variable at each iteration approaches a geometric series, the acceleration step

$$x_i^\star \approx \frac{x_i^{k-1} x_i^{k+1} - (x_i^k)^2}{x_i^{k-1} - 2x_i^k - x_i^{k+1}} \tag{2.20}$$

is taken. If Aitken's method is used on the full problem, then N equation solutions are required per iteration, and at each acceleration step, $4N$ divisions and $3N$ additions are necessary. When it is used on the torn problem, there are still N equation solutions to be found but the work at each acceleration is reduced to $4c$ divisions and $3c$ additions. Given that equation evaluation is more expensive than an arithmetic operation this is an insignificant saving in effort.

Newton Like Methods This class of numerical methods will be represented by the Newton Raphson method. This uses the function values at x^i and the partial derivatives at this point to find the value of x^{i+1}. If the functions to be

solved are of the form

$$f(x) = b \tag{2.21}$$

they are rewritten as

$$f(x) - b = 0 \tag{2.22}$$

and solved by a truncated Taylor expansion of equation 2.22. The Jacobian, J,

$$J = \{\frac{\partial f_i}{\partial x_k}\}_{i,k=1}^{i,k=N} \tag{2.23}$$

is required. The computational scheme required for the full case is

1. Evaluate $f(x^i) - b$
2. Evaluate $J = \nabla f$
3. Solve $Jx^{i+1} = -f(x^i)$ for x^{i+1}

If some of the variables are torn, this scheme becomes

1. Evaluate $x_{nc}^i = f_{nc}(x_c^i)$
2. Evaluate $f_c^r(x_{nc}^i(x_c^i),\ x_c^i)$
3. Evaluate $J_{nc} = \nabla_c\ f_{nc}$
4. Evaluate $J_c = \nabla_c\ f_c^r(x_{nc}^i(x_c^i), x_c^i)$
5. Solve $J_c x_c^{i+1} = -f_c^r(x_c^i)$ for x_c^{i+1}

Here x_{nc} is the set of dependent variables and x_c is the set of independent (*i.e.* tear) variables. $f_{nc}(x_c)$ is the set of equations used to solve explicitly for x_{nc},[2]

[2]**n.b.** These equations are rearrangements of the first $N - c$ equations.

$f_c^r(x_{nc}(x_c),\ x_c)$ is the set of reduced equations which is used to solve for the tear set and ∇_c is the gradient vector for these variables. J_{nc} is the Jacobian for the dependent equations and J_c is the Jacobian for the reduced system.

There are two ways in which each Jacobian may be calculated and these are considered in turn.

Analytical Jacobian In the full case N $(N+1)$ function evaluations are required for the first two steps and, as is shown in appendix A, $O(N^3)$ operations to solve the Jacobian equation for $x^{(i+1)}$. If the variables are torn, these requirements are reduced to c $(N-c)$ function evaluations for the chained derivatives and N $(c+1)$ function evaluations for the reduced equations Jacobian, and $O(c^3)$ operations to solve the reduced Jacobian equation for $x_c^{(i+1)}$. If $c \ll N$ this represents a considerable saving both in operations per iteration and storage requirement.

Numerical Jacobian In order to evaluate the Jacobian it is necessary to evaluate the relevant equation set at the current value of the variables, to perturb each in turn and then revaluate the equations before resetting the variable to its original value. Thus each element of the Jacobian is generated from

$$J_{ij} = \frac{f_i(x_k + \delta x_k) - f_i(x_k)}{\delta x_k} \tag{2.24}$$

where typically $\delta x_k = \epsilon x_k$ for some small value ϵ. If the Newton Raphson method is used on the full equation set this requires N $(N+1)$ function evaluations. If it is used on a torn system, however, $(N-c)$ $(c+1)$ function evaluations are required to calculate the chain rule derivatives and c $(c+1)$ are necessary for

the reduced Jacobian. This amounts to a total number of function evaluations of $N\ (c+1)$ which is a small saving over the full case.

Wegstein's Method Wegstein [99] developed the secant method for single equations, and his method has been generalised for the solution of multidimensional problems. In his method, the new value of x after the $(i+1)^{th}$ iteration is found from

$$\Delta x^i = -B^i f^i \tag{2.25}$$

where $\Delta x^i = x - x^{i+1}$, and the matrix B^i is found from

$$b^i_{jk} = \frac{x^{i+1}_k - x^{i-1}_k}{f_j(x^{i-1}_k)} \tag{2.26}$$

Whether this is applied to the full or the torn problem, N function evaluations are required per iteration. In the full case, however, N multiplications and $2N$ additions are required as well, whereas for a torn equation set this is reduced to c multiplications and $2c$ additions. This is a very small saving, but there is a much larger difference in the effort required to solve equation 2.25. For the full case, this requires $O(N^3)$ operations, but for the torn case only $O(c^3)$ are needed. If $c \ll N$ this may be a significant saving.

Thus, of all of the methods considered here, tearing the equations produces significant savings in computational effort at each iteration for only the secant method, for the Newton Raphson method and for Wegstein's method.

2.7 Summary

In this chapter it has been shown that whilst that no assurance of the existence of a unique solution to an equation set can be gained by an inspection of its structure, in the common case, some necessary conditions can be placed upon this. Next it was shown that, whatever the solution method, it was desirable to find an output set for the equations and the bounds on the number of such sets were established. In § 2.5 the equivalence of matrix, graph and equation partitioning was demonstrated in that it was shown that diagonal of a partitioned incidence matrix for some equation set $F(X)$ corresponded to the strong components of the digraph of the assigned equations; these in turn represent a sequence of equation sets which can be solved simultaneously. The effect of this partitioning on the amount of effort required to solve an equation set, both per iteration and on the number of iterations was discussed. Lastly, in § 2.6 it was shown that it is very hard to define conditions on the optimality of a tear set for a problem. Further, whilst an analysis of the geometric effects of tearing is not possible, it was shown that in many cases there was no significant reduction in the operations count per iteration for a torn system over the full case.

We proceed in the next chapter to discuss the most popular methods of selecting an output set and partitioning and decomposing equation sets.

The evil that men do lives after their lives, yet the good is oft interred with their bones

William Shakespeare, Julius Caeser

Chapter 3

Literature Review and Selection of Methods

3.1 Introduction

In the last chapter we discovered why some structural analysis of equation sets is necessary, and why other aspects of this phenomenon are desirable. In this chapter we discuss solutions which have been proposed for the problems raised in that chapter. In § 3.2 methods for output selection are described. There are multifarious formulations of this problem which draw on techniques such as graphical analysis and integer programming. In § 3.3 we turn our attention to the partitioning of incidence matrices. Here we demonstrate how the structure of these matrices determines the pattern of fill in which they experience, and it is shown that no deterministic algorithm exists which can predict a minimum for this phenomenon. Next methods for permuting the rows and columns of these matrices are described, and lastly we consider algorithms which manipulate them in different ways. In § 3.4 decomposition techniques are examined. These

range from arbitrary strategies, through integer programming methods to those which are based on a depth first search of the bipartite digraph which describes the matrix. There are some techniques which can be considered common to graph partitioning and decomposition. Those which select spikes may be used as tearing methods since they identify the effects of circuits within both directed and undirected graphs.

A brief summary of the conclusions drawn from each section is presented in § 3.5.

3.2 Choosing An Output Set

Consider an equation set R which involves the variables, X, and which is structurally non-singular. If $v \in X$ is a variable which appears in $u \in R$ then, given the values of all of the other variables in u, this equation can be solved for x. Thus u and v can be recorded as a pairing and u is said to be assigned to v. No ordering is defined on assignment and so, alternatively, v may be said to be assigned to u. A set, S, given by

$$S = \{(u_i, v_i)\} \quad i = 1, 2, \cdots, \ |R| \tag{3.1}$$

in which $u_i \neq u_j, i \neq j$ and $v_i \neq v_j, i \neq j$, is called a maximal assignment for the equation set; equivalently, S may be referred to as an 'output set' or a 'maximum transversal'. The first algorithm which appeared explicitly for the purpose of identifying an output set was presented by Steward [92]. He was concerned with identifying a single output set and then showing that this set could be used to generate all others; he showed too that where no such output set exists the

equation set is structurally singular. He presents his explanation in terms of the incidence matrix of the equation set, but it is clearer to consider the undirected, bipartite graph $\mathcal{G}(V, E)$ which represents it, and $\mathcal{D}(V, \acute{E})$, the directed bipartite graph which corresponds to a particular assignment of variables to equations.

The first step in his algorithm is an assignment of a variable, v, to an equation, u_k, whose node is adjacent to that of v in $\mathcal{G}(V, E)$. If this node already appears in some element, $(u_k, w_k) \in S$, as defined above, then the pair, (u_k, w_k), is removed from S and replaced by (u_k, v); an attempt is made now to assign a new equation node to w_k. Any such assignment may cause other pairs in S to be removed and replaced and the process continues until either an equation node, u_j, is encountered which is not in S, in which case the new assignment is added to this set, or there are no more equation nodes which are candidates for assignment to the current variable node, w_j. In this case each of this node's predecessors on the path is examined to see if it can be assigned to a free equation node. If this is possible S is perturbed in the usual way and a fresh search is made for the next variable node; if this is not possible then, since no assignment can be found which includes each variable, R must be structurally singular.

Having identified an output set, Steward shows that all others can be found by forming a symbolic version of the reachability matrix . This is defined to be the N^{th} power of $Adj(R)$, the incidence matrix for R, where $|R| = N$; the diagonal blocks of this matrix represent all circuits in $\mathcal{G}(V, E)$ which involve N edges. By defining the processes of directed path multiplication and addition he shows that in each of the i^{th} powers of $Adj(R)$, all paths in $\mathcal{G}$ which involve i edges are recorded and that each is recorded i times. His argument is that evaluating all of the loops of $\mathcal{G}$ in this way allows the generation of new output sets simply by

reassigning nodes around these loops, possibly recursively. This makes implicit use of theorem 2.1, which states that each of these reassignments must take place within a strong component of $\mathcal{D}(V, \dot{E})$.

Although each of these algorithms is correct neither is very good because each is algorithmically inefficient. In the first the lack of a look ahead facility may lead to a large waste of effort. Consider the situation when M assignments have been made and we wish to assign an equation to the $M + 1^{th}$ variable. In the worst case, all paths of length $l \leq M$ in $\mathcal{G}(V, E)$ which involve the M assigned variables may have to be searched before a new assignment is found. Whilst it is difficult to express the worst case algorithmic complexity for this method, it is certainly very high. In the second it is very expensive to calculate powers of the incidence matrix, even when it is stored in packed form.

A better algorithm based on a depth first search of $\mathcal{G}(V, E)$ and which includes a look ahead facility is that due to Duff [23]. This author defines a cheap assignment to be an assignment which is made without resort to a path search. In terms of the incidence matrix of $\mathcal{G}(V, E)$ this corresponds to assigning to the row, i, the first column, j, which intersects with it and which is not in the present assignment. Staying with this representation of the equation set, the algorithm starts by making as many consecutive cheap assignments as possible. Whenever this process fails for some row a path search is started, even if a cheap assignment is possible for some row later in the matrix.

Let such a row be i_0. The search starts by finding the first non-zero in this row; this is in column j_1 and it has been assigned to row i_1. Row i_1 is searched now and if it contains a free non-zero j', the assignment (i_1, j_1) is removed from this

set and replaced by the pair of assignments, (i_0, j_1) and (i_1, j'), and the search is restarted from the next free row. If i_1 had contained no free column then the first column with which it intersects j_2, $j_2 \neq j_1$, would have been placed on the path and the search continued. This process of extending the path is the depth first search and the search for a free column in each row is the look ahead facility. If during the search a column has no more candidate rows, the search backtracks to the previous row.

Duff [23] interprets this search by reference to an obscure form of a signal flowgraph. A much clearer interpretation is apparent if one treats it as an attempt to establish a maximum matching, $\overline{M}$ [36], in an undirected, bipartite graph where the vertex partitions, V_y and V_x, correspond to equations (rows) and variables (columns) respectively. Recall that a matching in a graph is a subset of its edges such that no vertex appears in more than one edge. A matching of maximum size for a graph is called a maximum cardinality matching; if each vertex in the graph is incident on one of the edges in such a matching it is said to be complete. Any vertex which is not an endpoint of some edge in the matching is said to be free and a path of odd length between two free vertices in the graph, such that there is no other free vertex on it, is termed an augmenting path. If this path is of length $l > 1$ then the edges of which it consists are alternatively in the current matching and outwith it. Let $\mathcal{P}$ be such a path, $\mathcal{A}$ the set of edges which it contains and $M_0 \subseteq \mathcal{A}$ be the edges in $\mathcal{A}$ which are also in M, the current matching. Since both terminal vertices in $\mathcal{P}$ are free, $|\mathcal{A} - M_0| = |M_0| + 1$, and thus the size of the matching may be increased by one by removing each member of M_0 from M and adding each member of $\mathcal{A} - M_0$. In a finite graph, if no augmenting path exists then the current matching is maximum.

The search starts by establishing the assignments (ν_i, ν_j), $\nu_i \in V_y, \nu_j \in V_x, (\nu_i, \nu_j) \notin M$ the current matching, until no such assignment can be made for some vertex $v_k \in V_y$. The search continues along a path, P, as described above and the lookahead corresponds to looking for a free vertex, $v' \in V_x$, which is adjacent to the vertex, $v_m \in V_y$, at the head of P. As Duff [23] indicates, it is difficult to define a worst case time complexity for this search, but it would appear to be $O(n\tau)$ where there are n vertices in $\mathcal{G}(V, E)$ and τ edges.

Westerberg and Edie [102], [103] presented an entirely different approach to determining an output set for the solution of linear equations. They argued that it is not only the structural form of a matrix which is important, but also the algebraic and numerical properties of the equations which it is used to represent. To this end they presented two strategies for improving the convergence characteristics of an equation set which is to be solved by successive substitution; they claimed that any strategy which improves the convergence of successive substitution is likely to improve the convergence of any other numerical method. The method of successive substitution will converge a set of linear equations if and only if the largest eigenvalue of its iteration matrix is less than one. If the equations to be solved are

$$Ax = f \tag{3.2}$$

then, using $A = D - B$, where D is a diagonal matrix with the same entries as the diagonal of A, the method of successive substitution finds x by

$$x = D^{-1}Bx + D^{-1}f \tag{3.3}$$

and $D^{-1}Bx$ is the iteration matrix. These authors show that this value can be minimised either by minimising the maximum row sum in this matrix or maximising the product of the diagonal coefficients. Either of these goals may be

achieved by the application of dynamic programming techniques and an implicit enumeration method is presented for each which cuts down the amount of search required.

These techniques can be extended to deal with non-linear equations if an iterative solution procedure is used and the ordering method is applied to the Jacobian. The authors suggest that the first derivatives of the equations be used and that the output set be chosen before the first iteration. Should the solution vector change appreciably, then the output set ought to be redetermined. Given the amount of effort required in solving a dynamic program to determine each output set, and the crudeness of the measure of optimality, this seems unlikely to be of any real benefit.

Sargent [82] proposed that the selection of an output set could be posed as the set partitioning problem:

$$\begin{aligned} max \quad & \sum_{j=1}^{j=N} \omega_j x_j \\ \text{s.t.} \quad & \sum_{j=1}^{j=\tau} C_{ij} x_j = 1, \quad i = 1, 2, \cdots, 2N \\ & x_j = 0 \text{ or } 1, \quad j = 1, 2, \cdots, \tau \end{aligned} \tag{3.4}$$

where C is the node-arc incidence matrix for the equation set[1], $\{x_j\}$ is the set of variables and equations and ω_j is a weight assigned to the j^{th} arc; this weight reflects the desirability of adding the pairing corresponding to the endpoints of the j^{th} arc to the current matching. In this formulation, each arc in the bipartite graph which represents the equation set is assigned a weight and the maximum sum of N of these arc weights is chosen within the constraint that each variable

[1]In this matrix the rows correspond to nodes and the columns to arcs in the bipartite graph. The column for edge e has exactly two entries, and these are in the rows which represent its termini.

and equation is an an endpoint of exactly one arc. Sargent is not explicit about the details of this formulation, but he suggests that if one wishes only to identify one output set then equations 3.4 might be solved using either the algorithm of Edmonds [27] or Hopcroft and Karp [46] but that if one wishes the optimal solution then that of Edmonds and Johnson [28] should be used instead. The first of these algorithms has a worst case time complexity of $O(N|E|)$, where there are E edges in the graph, and the second has one of $O(N^{\frac{5}{2}})$. The third is less efficient yet. Even refers to a report by Gabow [33] in which its complexity is given as $O(N^3)$ and so, since assigning meaningful values to the weights can be very difficult, it would appear that there is very little point in finding the optimal solution to this problem. Further, as will be shown on page 77, there is a more efficient formulation of the output set problem.

Before describing this formulation it is worth noting that another approach which involves the ascription of weights to the arcs is the stable marriage problem. Here the point is to find a one to one correspondence between two disjoint vertex sets such that there are no two vertices i and j which are assigned to other nodes but which have a stronger mutual attraction. Gale and Shapely [34] have presented an algorithm which finds a solution to this problem in $O(N^2)$ time where there are N vertices in each set. Irving [48] has shown that determining the number of solutions to this problem for any value of N is NP-complete and so, in the absence of any polynomial time algorithms, it is likely that determining the optimal solution for this formulation is also NP-complete. A further problem with this technique is that it is difficult to define what one means by the optimal solution. Let one of the vertex sets be labelled 'men' and the other 'women' and let a good solution be one in which one of the vertex sets has its preferences satisfied to a maximal degree. In general an assignment which is man optimal

will not be woman optimal and vice versa and so some form of compromise must be reached. In terms of equation solving this means that an assignment which matches each variable to the equation which is most easily solved for it, within the constraints of the problem, is unlikely to match each equation to the variable within it for which it is most easily solved. Thus an optimal solution, however it is defined, lies somewhere between these two extremes. Even if one were able to define the relevant optimality criteria there is no guarantee that this would have any real meaning since it ignores the values of the variables. This is an unpromising approach and, given the same edge weights as in Sargents formulation [82], it gives a suboptimal solution. It would be acceptable only if it were much more efficient, but it may be useful for providing a starting point for the set partitioning problem.

Paterson [69] has provided a possible means of circumventing the problem of assigning weights to the edges in the graph. He restricted his work to the solution of a single equation in a single variable but his results may be extended to cover the multidimensional case. His argument is that one ought to rerearrange a non-linear equation so that it can be rendered nearly linear by a suitable change of variable, *e.g.* by replacing a squared term by a new variable. This is desirable because those numerical methods which have superlinear convergence use derivative information which approximates a curved gradient by a straight line. A second desirable condition is that the right hand side of the new equation should be a weak function of the variable on the left hand side so that the absolute value of the gradient will be less than one . The desirability of this condition arises from the assurance of the convergence by successive substitution of such an equation.

This author suggests that when solving an equation by a Newton type method one ought to rearrange it so that the equation being solved is a difference between such a right and left hand side. Thus if the original equation $f(x) = 0$ is rearranged so that $x = f(x)$ is a good rearrangement for solution by successive substitution, the convergence characteristics of Newton's method for $x - f(x) = 0$ ought to be better than those for $f(x) = 0$. Paterson [70] extends this idea to providing good rearrangements and starting guesses for equations by identifying the dominant term in an equation if one exists. Having identified this term, he gets a good starting guess for the iteration by approximating the equation to this term and solving the approximation analytically. The original equation is then rearranged so that, after a change of variable, the dominant term is now the subject of the equation.

Paterson's argument [69] is that these techniques work because they satisfy a sufficient condition for convergence, and they perform better than the method of successive substitution (MSS) early on in the iteration, and thus better overall. As he points out, a sufficient condition for the convergence of this technique for the solution of some equation $f(x)$ is that $\mid f(x^*) \mid < 1$, where x^* is the desired solution. Since the value of x^* is unknown, he relaxes this condition to hold on the value of x^0, the initial estimate of the solution. This is his justification for making the right hand side of an equation a weak function of the left hand side. Whilst there is a plausible argument, supported by experience, that the use of Paterson's observations are likely to improve the convergence characteristics of an equation set, it is not true to say that he has defined a condition which is sufficient for this.

Paterson's techniques extend to cover the multidimensional case in a natural

way. Here an equation is rearranged for some variable within it which does not yet appear in an assignment and which appears in a term which can be used to maximise the linearity of the rearranged equation. It is clear now how Paterson's work relates to formulations of the assignment problem in which edges are weighted; his analysis of each equation can be used to assign the weightings for the edges between variables and equations. Such a weighting could be assigned *a priori* or reviewed once every few iterations. This is likely to be extremely expensive, however, because each variable may occur in more than one term in each equation, and so many rearrangements and approximate solutions would be required to calculate these weights. Ascribing these weights would embody the majority of the effort required to produce an assignment. Since the cost of solving a stable marriage problem or a dynamic program, probably suboptimally, provides the balance, this technique is unlikely to be of practical use.

The most efficient formulation of the output set problem where no weights are taken into account is that of modelling it as a flow network problem. Each edge, e, of the bipartite digraph $\mathcal{D}(V, \acute{E})$ is assigned a capacity, $c(e)$, which is the largest amount of flow allowed through it. The purpose of the algorithms presented here is to find the maximum possible flow from one partition to the other; the material is assumed to flow from an imaginary source, which is connected to each of the nodes in one of the partitions, into an imaginary sink which is attached to each of the nodes in the other. Prior to describing the formulation in detail we require the following definitions.

A flow function, f, is an assignment of a number, $f(e_i)$, to each edge, e_i, in a graph. Clearly

$$0 \leq f(e_i) \leq c(e_i) \tag{3.5}$$

The total flow, $\mathcal{F}$, through the graph is the net flow from the source to the sink. A network, $\mathcal{N}$, is a directed graph which has a source and a sink and for which every edge, e_i in $\mathcal{N}$, has a capacity, $c(e_i)$. If initially

$$\left.\begin{aligned} f(e_i) &= 0 \\ c(e_i) &= 1 \end{aligned}\right\} \forall\, i \tag{3.6}$$

if the toal flow through each node other than the source and the sink is restricted to unity; and if only integer increments are allowed in $f(e_i)$, then $\mathcal{N}$ is called a zero-one network; this is the type of network which is of interest to us. An example of such a network is shown in figure 3.1. An edge e_i is said to be useful

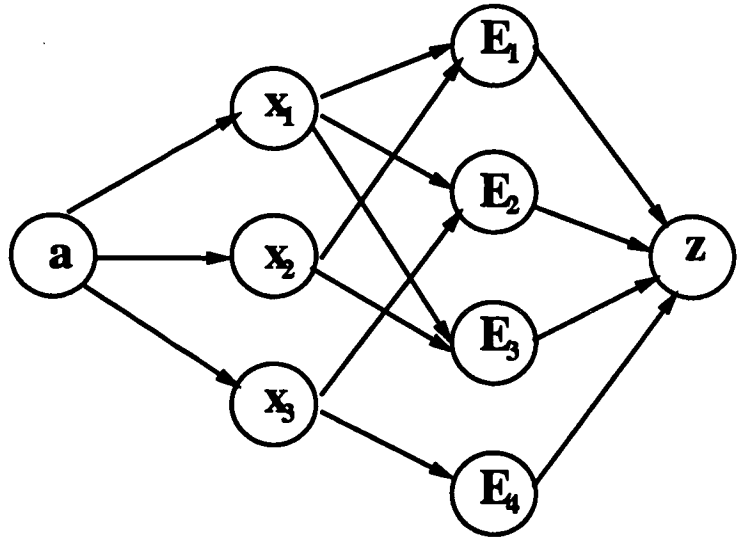

Figure 3.1: An example of a Network

if it connects two nodes, u and v, where u is closer to the source, *i.e.* there is a shorter path from the sink to u than from the sink to v, and either

$$e_i = u \rightarrow v \;\; and \;\; f(e_i) < c(e_i) \tag{3.7}$$

or

$$e_i = u \leftarrow v \;\; and \;\; 0 < f(e_i) \tag{3.8}$$

because in either case the net flow of material from the source can be increased by forcing flow through e_i towards the relevant bound. A proof that flow augmenting techniques can be used to solve the output set selection problem is deferred until § 4.2.

Possibly the best known algorithm for maximising network flow was provided by Ford and Fulkerson [31]. At each stage the search for an augmenting path starts at the source and a vertex, v_1, is sought through which the flow is submaximal. A similar vertex, v_2, which is adjacent to v_1 is sought and the process continues until the sink is reached; at this point the path has been found and flow is increased along it by the maximum amount possible which does not break constraints 3.5. When no such augmenting path exists the flow is maximal. This algorithm may fail in the general case if $c(e_i)$ is allowed to be irrational for any $e_i \in E$; hence the need to constrain $c(e_i)$ to integral values. The nature of this algorithm has been reviewed by Even [30], and he refers to a breadth first search amendment which guarantees that the algorithm will terminate in $O(|V|^3|E|)$ steps even when $c(e_i)$ is allowed to be irrational.

A much better method is that due to Dinic [20]. This algorithm uses a breadth first search through a network, $\mathcal{N}$, which changes each time that an augmenting path has been found. One can show that this algorithm must terminate and that it must do so after $O(N^2\tau^{\frac{1}{2}})$ steps, where there are N nodes and τ arcs in the network. Even [30] has proved that for a zero-one network this bound is reduced to $O(\tau^{\frac{3}{2}})$. This algorithm is described fully in § 4.2 and so it will not be discussed here, save to say that it appears to be the most efficient method for determining an output set for a set of equations.

It has been shown here that Steward's [92] seminal algorithms for establishing an output set and generating all others from it are highly inefficient, and that they have been superseded by most of their modern competitors. Further, attempts to define an optimal output set have not produced criteria which are both meaningful and efficiently established. Westerberg and Edie's, [102] and [103], methods for minimising the maximum eigenvalue of an iteration matrix are meaningful but inefficient; Sargent's [82] set partitioning formulation provides an optimal solution in $O(N|E|)$ time, but he does not present any method of assigning weights to arcs. Regarding assignment as an instance of the stable marriage problem guarantees an answer where the equation set is structurally non-singular, but it is both difficult and costly to define an optimal solution, even when Paterson's methods, [69], [70], are used. Duff's depth first search algorithm with a lookahead facility identifies a maximal assignment in $O(N|E|)$ time, and it is easier to implement than Sargent's integer program. The most efficient approach which has appeared, however, is to treat it as a maximal flow problem in a zero-one network which is to be solved by an the application of Dinic's method [20].

3.3 Partitioning Matrices

In this section we will discuss not only methods of partitioning matrices but also ways of permuting the rows and columns within diagonal blocks. When we are dealing with the solution of non-linear equations, rows and columns are permuted within blocks so that they have bordered triangular form (see figure 2.11). The variables which correspond to the right hand border are then torn; these techniques will be described in § 3.4. When linear equations are being

solved, blocks are ordered internally so that fill-in is controlled during Gaussian Elimination.

A spike is a column which has non-zero entries above the diagonal. At first sight, since fill-in can occur only in spikes, one might believe that minimising the number of spikes would minimise the fill-in in a matrix. Consider, however, the two matrices in figures 3.2(a) and (b). These are symmetric permutations of one

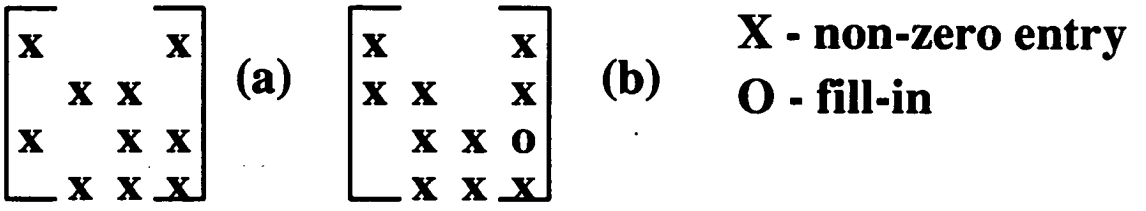

Figure 3.2: Two permutations of an Irreducible Matrix

another and figure 3.2(a) has two spikes whereas figure 3.2(b) has only one. If Gaussian Elimination were applied to the matrices then no fill-in would occur in the first matrix whereas one entry would fill in the second, despite the fact that it has one less spike. Prior to a discussion of how fill-in can occur, it is necessary to establish the relationship between different permutations of the same matrix.

Let A be any matrix of order, N, and let P be some permutation matrix of the same order such that $P_{ij} = 0$ *or* $1, i, j = 1, 2, \cdots, N$, and there is at least one non-zero entry in each row and column. The matrix

$$A_1 = PA \tag{3.9}$$

has the same elements as A but its rows appear in a different order.

Postmultiplying A_1 by the transpose of P to give

$$A_2 = A_1 P^t \quad (= PAP^t) \tag{3.10}$$

permutes the columns of A_1 in the same way as its rows. As shown by theorem 2.5 the diagonal blocks of A, A_1 and A_2 are permutations of each other and hence all matrices

$$\overline{A} = RAR^t \tag{3.11}$$

such that $\overline{A}$ is block lower diagonal, form an equivalence class for all permutation matrices R.

If A_1 is postmultiplied by some permutation matrix $Q \neq P^t$ to give

$$A_3 = PAQ \tag{3.12}$$

then the columns of A_3 are permuted in a different way to the rows of A. If A had non-zeros in every diagonal position to begin with then A_3 belongs to the same equivalence class as RAR^t, Duff [22]. The graphical interpretation of equation 3.11 is that it reorders the nodes in the digraph of A whereas equation 3.12 reorders the nodes and reorients some of its edges.

3.3.1 A Characterisation of Matrix Partitioning

Rose and Bunch [79] showed that permuting an irreducible matrix never saves arithmetic operations, regardless of whether this is performed symmetrically or asymmetrically, although it can lower storage requirements. In order to demonstrate this, they cited the solution by Gaussian Elimination of the

equations

$$Mx = k \tag{3.13}$$

where M is an $N \times N$ coefficient matrix and x and k are $1 \times N$ vectors. Performing a Gaussian Elimination on the first m rows of M can be regarded as finding a partial LU decomposition of this matrix, and this can be written as

$$M = \begin{bmatrix} L_1 & 0 \\ R_1 U_1^{-1} & I \end{bmatrix} \begin{bmatrix} U_1 & L_1^{-1} \\ 0 & \Delta \end{bmatrix} \tag{3.14}$$

where L_1 and U_1 are $m \times m$ matrices, R is an $N - m \times m$ matrix and C is an $m \times N - m$ matrix. Since M is irreducible, so too is every permutation of it and hence R can never be the zero matrix. If the graph is not strongly connected, then $R = 0$ is possible, and both storage and arithmetic requirements may be reduced. Rose [78] defined a perfect elimination undirected graph to be one whose nodes are ordered so that, on elimination of some node, x_i, no new edges have to be added to the graph so that all paths of length, $l > 1$, which pass through x_i in the original graph $\mathcal{G}(V, E)$ become paths of length $l - 1$ in the new graph, $\mathcal{G}_{x_i}$. Not all graphs may be ordered in this way and not every ordering of one which can is a perfect elimination ordering.

This definition is important in the study of the solution of equation sets. Let the rows and columns of the $N \times N$ symmetric incidence matrix, $A(\mathcal{G})$, be ordered in the same way as the nodes of $\mathcal{G}(V, E)$. Then the elimination of the i^{th} node from $\mathcal{G}(V, E)$ corresponds to pivoting on the i^{th} row and column of $A(\mathcal{G})$. In general, pivoting leads to fill-in, and this corresponds to adding new edges to the reduced graph. In order to relate this fill-in to Gaussian Elimination, Rose [78] made use of the following definitions.

The deficiency of $v_i \in V$, $D(v_i)$, is defined as

$$D(v_i) = \{(v_j, v_k) \mid v_j, v_k \in Adj(v_i),\ v_j \notin Adj(v_k),\ v_k \notin Adj(v_j)\} \qquad (3.15)$$

i.e. the set of edges whose addition to E would make the vertex subset $Adj(v_i) \cup \{v_i\}$ a clique. The elimination graph of v_k in $\mathcal{G}$ is $\mathcal{G}_{v_k}(V - \bigcup_{j=1}^{j=k} v_k,\ \overline{E})$, where

$$\overline{E} = (E - \{(v_j, v_i) \mid v_i \in \bigcup_{j=1}^{j=k} Adj(v_j)\}) \bigcup_{j=1}^{j=k} D(v_j) \qquad (3.16)$$

which is the graph obtained by deleting the vertex v_k from the $(k-1)^{th}$ reduced graph $\mathcal{G}_{v_{k-1}}$ and adding those edges in its deficiency. In figure 3.3, the graphs of figures 3.2 (a) and (b), the first graph has a null deficiency, whereas the second has $D(E_3) = \{x_4\}$. The arc (E_3, x_4), which is shown as a dotted line, is added to the edge set when E_3 is eliminated. According to Rose, the (possibly filled) submatrix, $\tilde{A}(\mathcal{G})$, which results from pivoting on the k^{th} row and column of A is the incidence matrix of the graph $\mathcal{G}_{v_k}$. To see this one need note only that pivoting on this vertex involves the deletion of each entry in the k^{th} column of A which lies beneath the k^{th} row, and the modification of the non-zero entries in each affected row which lies to the right of the k^{th} column. A row is affected if and only if it corresponds to a node in the adjacency set of v_k; each entry in the i^{th} row is affected if and only if it corresponds to a node in $Adj(v_k)$ or $Adj(v_i)$. If some entry, (i, j), is affected such that $v_i \in Adj(v_k)$ but $v_j \notin Adj(v_k)$, then a new non-zero entry is made in A. This corresponds to the creation of a new arc in $\mathcal{G}_{v_k}$ between node v_i and node v_j; no such new arc results from the case $v_j \in Adj(v_i)$, $v_j \notin Adj(v_k)$. Each new arc is a member of $D(v_k)$ and it is easy to see that each member of $D(v_k)$ contributes a new arc to $\mathcal{G}_{v_k}$. Thus $\mathcal{G}_{v_k}$ is the graph of the submatrix of A which results from pivoting on the k^{th} diagonal element of A.

It follows that if F is the set of new arcs added to $\mathcal{G}$ as each node is eliminated

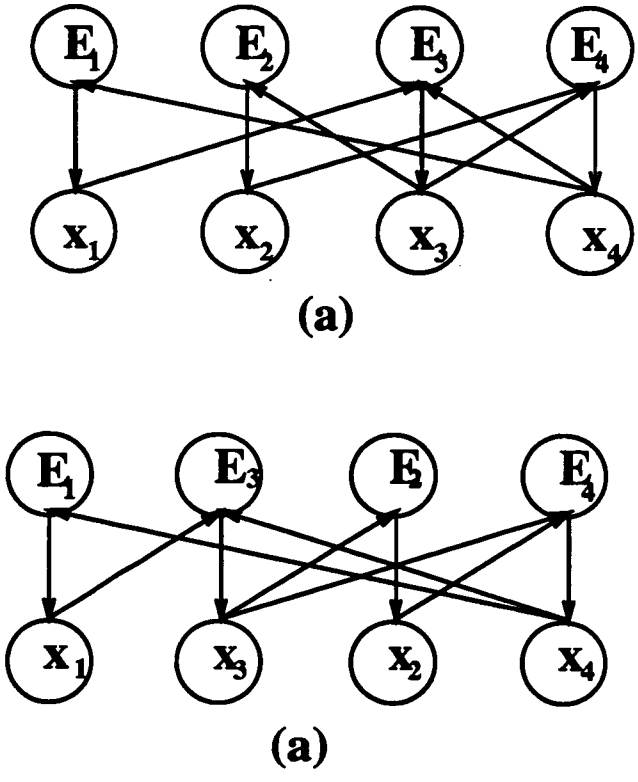

Figure 3.3: Graphs for the Matrices of figure 3.2

in turn, *i.e.* $F = \bigcup_{j=1}^{j=N} D(v_j)$ then F corresponds exactly to the set of filled entries which occur during vertex elimination on A. Further, $\mathcal{G}^F(V,\ E \cup F)$ is the graph of the matrix $L + L^T$, where L is the Cholesky factor of A^2 [51]. It is important to minimise the size of F so as to minimise both the storage requirements and the number of arithmetic operations necessary at each iteration. Ideally $|F| = 0$ is sought, and Rose shows that if this is to be achieved, then it is necessary for $\mathcal{G}(V, E)$ to be the transitive closure of itself, *i.e.*

$$(v_i,\ v_j) \in E \text{ and } (v_i,\ v_k) \in E \Rightarrow (v_j,\ v_k) \in E \tag{3.17}$$

Any graph which displays this property is said to be chordal [11]. This term is

[2] *n.b.* This refers to the logical Cholesky factorisation - no numerical values are assumed. This generalises Rose's argument to the solution of linear and non-linear equations.

used because each path in the graph of three vertices has a subpath between its termini.

Haskins and Rose [39] showed that fill-in may occur in the same way in unsymmetric matrix by relating this to vertex elimination in digraphs. They demonstrated that if there is a path in a graph $\mathcal{D}(V, \acute{E})$ from some vertex ν_x to another vertex ν_y which passes through one or more vertices which are ordered before ν_x and ν_y, then if $(\nu_x, \nu_y) \notin E$, this edge fills in when ν_z, the highest ordered vertex on this path such that it is ordered before ν_x and ν_y, is eliminated; the edge is directed in the same way as the path. More formally, if there exists a bijection

$$\alpha : V \leftrightarrow \{1, 2, \cdots, |V|\} \tag{3.18}$$

which orders the nodes in $\mathcal{G}^d$, then for any path, p,

$$p = \nu_x, \nu_{i_1}, \nu_{i_2}, \cdots, \nu_{i_m}, \nu_y \tag{3.19}$$

which contains at least one node ν_{i_j} such that

$$\alpha^{-1}(\nu_{i_j}) < min(\alpha^{-1}(\nu_x),\ \alpha^{-1}(\nu_y)) \tag{3.20}$$

then either $(\nu_x,\ \nu_y) \in E$ or $\mathcal{G}(V, E)$ is not a perfect elimination digraph. They extend their analysis by providing three necessary conditions for the perfect elimination condition on $\mathcal{G}(V, E)$:

1. $\forall\ \nu_i, \nu_j \in V$ at least one of these vertices, say ν_i is such that $\forall\ \nu_k, \nu_l \in V$ which separate ν_i and ν_j, ν_i does not separate ν_k and ν_l.

2. $\forall\ \nu_i, \nu_j \in V$ at least one of these vertices, say ν_i is such that $\forall\ \nu_k, \nu_l \in V$ which separate ν_i and ν_j, every set Γ of $n > 2$ vertices contains a subset

Ψ of $n-1$ vertices such that any path from ν_k to ν_l through ν_i whose elements are exactly those of Γ has a subpath from ν_k to ν_l whose elements are exactly those of Ψ.

3. For any set Γ of $n > 2$ vertices there exists a subset Ψ of $n-1$ vertices such that any cycle on Γ has a cycle on Ψ.

The authors conjectured that the first two of these conditions might be sufficient to ensure that $\mathcal{D}(V, \acute{E})$ is a perfect elimination digraph, but Kleitman [50] showed that this is not the case. The third condition is tantamount to saying that $\mathcal{G}(V, E)$ must be chordal.

Rose and Tarjan [80] extended these concepts and produced an algorithm which computes the fill-in for any ordering, and one which will find a perfect elimination ordering for a digraph should one exist; each of these algorithms can be executed in $O(N\tau)$ time, where there are N nodes and τ arcs in the digraph. They also presented an algorithm which, starting from any fill set, F, will reduce it and reorder the digraph until it finds a minimal fill set, F_0; this algorithm works in $O(N^2(\tau + |F|))$ time. More theoretically, they showed that since there is a polynomial transformation which converts the Satisfiability Theorem of Calculus into the problem of computing the minimum fill-in for $\mathcal{D}(V, \acute{E})$, the latter problem is NP-complete. Yannakakis [106] has provided a similar proof for undirected graphs, based on Berge's observation [11] that any perfect elimination graph is chordal. The problem of computing the minimum fill-in may be formulated as a calculation of the minimum number of edges which must be added to $\mathcal{G}(V, E)$ in order to make it chordal. He shows that the NP-complete Optimal Linear

Rearrangement Problem is a reduction of this task[3].

Schreiber [84] extended the analysis of vertex ordering in undirected graphs by examining the structure of the graph which corresponds to the Cholesky factorisation of a symmetric matrix A. He defined $col(j)$ and $next(j)$ for the j^{th} vertex to be

$$col(j) = \{i > j \mid l_{ij} \neq 0\} \tag{3.21}$$

$$next(j) = min\{k \mid k \in col(j)\} \tag{3.22}$$

Obviously $col(j)$ is the set of nodes ordered after j to which it is connected by an arc in the filled graph, and $next(j)$ is the lowest numbered such node. Schreiber shows that, as a direct consequence of these definitions,

$$\begin{aligned} col(k) &\subseteq col(next(k)) \cup \{next(k)\} \\ col(n) &= \emptyset \end{aligned} \tag{3.23}$$

He uses these definitions to form the elimination tree, $T(V,\ N(L))$, for the filled graph, where

$$N(L) = \{(j, next(j)) \in E \mid 1 \leq j \leq n-1\} \tag{3.24}$$

This is an ordered tree rooted at v_n, the last node in the ordering. If $row(j)$ is defined as

$$row(j) = \{k < j \mid l_{jk} \neq 0\},\quad 1 < j \leq n \tag{3.25}$$

i.e. the set of vertices whose removal effects the j^{th} node, then it can be seen that $T_{row(j)\cup\{j\}}$ is an ordered tree rooted at node j. Further, $col(j)$ is the set of nodes on the path from the j^{th} node in $T(V,\ N(L))$ to the root of this tree. From this it

[3]An arrangement of the nodes in a graph $\mathcal{G}(V, E)$ is an ordering π of the nodes within it. With each edge $e = (\nu, \omega) \in E$ in this graph is associated the value $\delta(e, \pi) = (\pi^{-1}(\nu) - (\pi^{-1}(\omega))$, and the cost of the arrangement is defined as $c(\pi) = \sum e \in E\delta(e, \pi)$. The Optimal Linear Rearrangement problem is the question "For an integer k, is there an arrangement of the nodes in $\mathcal{G}(V, E)$ such that its cost $c(\pi) \leq k$?".

can be seen that orderings which minimise the depth and maximise the breadth of the elimination tree tend to minimise the fill-in in the incidence matrix.

It is important to note that it is the structure of the elimination tree which determines the fill-in during vertex elimination, not the number of spikes in the incidence matrix. To show this Liu [58] followed the same line of reasoning as Schreiber [84], and he demonstrated that the fill-in in a graph can be characterised by the leaf nodes of its elimination tree. This result follows from the proof of a theorem which states that vertex v_j is a leaf in the row subtree rooted at v_i if and only if $(v_i,\ v_j) \in E$ and there is no descendent of v_j, v_k, such that $(v_i,\ v_k) \in E$. The same author [60] showed that the set of orderings which preserve the order of the nodes in $T(V,\ N(L))$ is a subset of the set of orderings which preserve the set of filled edges; this, in turn, is a subset of the orderings which preserve the number of edges added to $\mathcal{G}(V, E)$. He uses this reasoning to show how sparsity can be maintained when reordering some of the vertices in $\mathcal{G}(V, E)$.

No characterisation of directed graphs in terms of an elimination tree has appeared as yet, but some progress in this direction has been made. Aho *et al.* [1] define the transitive reduction of a digraph to be the smallest graph $\mathcal{D}_1{}^t$ which has the same transitive closure as $\mathcal{D}(V, \acute{E})$. $\mathcal{D}_1{}^t$ need not be a subgraph of $\mathcal{D}(V, \acute{E})$, but it has the same number of nodes and its strong components, each of which is a simple cycle, are comprised of the same vertices as those of the larger graph; if there are one or more arcs between strong components in $\mathcal{D}(V, \acute{E})$, these are represented by a single arc in $\mathcal{D}_1{}^t$. Should $\mathcal{D}(V, \acute{E})$ be acyclic then $\mathcal{D}_1{}^t$ is unique. Otherwise there will be more than one transitive reduction of $\mathcal{D}(V, \acute{E})$ and the relationship between the transitive reduction and the transitive closure of $\mathcal{D}(V, \acute{E})$ is the same as the relationship between the leaves of $T(V,\ N(L))$ and the

structure of the filled undirected graph from which it it constructed. Sahni [81] defines the minimal equivalent digraph of a digraph, $\mathcal{D}(V, \acute{E})$ to be its minimal subgraph $\mathcal{D}_2$ which has the same transitive closure as $\mathcal{D}(V, \acute{E})$. He shows that finding this subgraph is an NP-complete problem.

3.3.2 Symmetric Permutations

Harary [38] presented a technique for partitioning the incidence matrix which uses the reachability matrix for a graph. He uses a slightly different definition of this matrix to that given on page 69. In his terms, this matrix is the k^{th} power of the incidence matrix and its $(i,j)^{th}$ element is non-zero if there is a path of length l, $l \leq k$ from node i to node j. If the incidence matrix of $\mathcal{D}$ has rank N then the $(N-1)^{th}$ reachability matrix contains all of the paths which exist within the graph. Each strong component can be found by checking along each row i to see if for each non-zero intersection with a column j, (j,i) is also non-zero; the set of all such non-zero entries defines the set of nodes which appear in the same strong component as i. Having deleted each row and column which corresponds to this strong component the search can continue; **n.b.** this does not order the strong components in any meaningful way. In the worst case, *i.e.* each node is in a different strong component, $\frac{N(N-1)}{2}$ checks are necessary to identify them and, if no packed form is used, $(N-1)N^3$ multiplications are necessary in order to compute the reachability matrix.

A similar but different definition of the reachability matrix was used by Himmleblau [43]. This author defined the non-zero entries of the k^{th} power of

the incidence matrix with zeros on the diagonal to correspond to node pairs such that there is a path of exactly length k between these nodes. The reachability matrix is then the summation of each of these matrices from 1 to $N-1$; the final matrix has the same form as that of Harary [38] but Himmleblau defined Boolean multiplication and addition differently. If R^* is the above mentioned sum then the set of non-zero entries in the i^{th} row of $R^*(R^*)^t$ contains all of the nodes which are in the same strong component as i. Once again this does not order the strong components of the graph.

Steward's algorithm [93] begins by finding a maximum transversal of A, the incidence matrix of the equation set, and forming $\mathcal{H}$, the signal flowgraph of the digraph based on the 'equation' nodes which represents the transversal. All of the sources for this flowgraph are eliminated, although none of the sinks is, and then its loops are identified by a depth first search. Not all of the loops are identified explicitly, but node j is collapsed into the supernode I if it is in a loop with any vertex $k \in I$ (a supernode is simply a loop which is treated as a node). This process is repeated until no new loops are found and the stack is then popped with each supernode containing a strong component of $\mathcal{H}$. As we will see in § 5.2 these are also the strong components of $\mathcal{D}(V, \acute{E})$.

None of these algorithms is very efficient because the first two require several powers of the incidence matrix to be evaluated and the second restarts each search for a loop from the start of the graph. Perhaps the most popular method is that due to Walker and Tinney [97], which Rose [78] called the minimum degree ordering. This algorithm was developed for use with symmetric matrices and it selects as the next node to be ordered that which has the lowest degree in the current reduced graph; **n.b.** this is a symmetric version of Markowitz's

[61] algorithm. Many authors have addressed themselves to improving the performance of this technique and their efforts are reviewed by George and Liu [35].

3.3.3 Asymmetric Permutations

Sargent and Westerberg [83] addressed the problem of partitioning within the context of precedence ordering of the calculations in a process flowsheet. Implicit in their approach is the assignment of a direction for each arc in the graph. This is implicit because there is a natural direction associated with an arc in a digraph which represents a chemical process, namely the direction of material flow. Therefore, prior to use of this algorithm for ordering equation sets, a search for a maximum transversal is necessary. They proposed a depth first search (DFS) algorithm which selects an arbitrary start vertex and searches backwards along the edges incident upon it in order to identify cycles of the digraph. When a loop is encountered the nodes associated with it are grouped together and treated as a single node; any edge which was incident upon one of the constituent nodes is held to be incident upon the supernode and likewise those edges incident from any of these vertices is incident from the group. Having encountered and formed a supernode the search is continued as before. Should a new node be in a loop with a supernode already on the stack then those nodes are merged, along with any others between them on the stack.

If at some point in the search all of the incoming edges for a node have been searched and it is found to be in no cycle with any other node then this node is

popped from the stack (it must be at the top) and added to the list of strong components. This is the case regardless of whether the vertex is simple or a supernode. Should an edge from such a vertex to a node on the stack be identified later no action is taken since such a path implies the existence only of a path, not a circuit. These authors seek to permute the rows and columns within the blocks of the incidence matrix which correspond to these strong components so that they are in bordered block diagonal form. The borders of these blocks are formed by minimising the weight of the spikes in each block using a dynamic programming technique similar to that used by Westerberg and Edie [102]. The amount of search within each block is minimised by the use of graph reduction and an implicit enumeration technique.

Christensen and Rudd [16] proposed a similar scheme to that above, but they allowed nodes to be permuted to the end of a sequence as well as to the beginning. They too proposed a method of node merging to reduce the size of the digraph. Forder and Hutchison [32] took a similar approach, but they enumerated all of the cycles in the graph by a depth first search, and employed a complicated flagging system in order to identify the first node in a strong component on the stack. The blocks of the incidence matrix are generated in reverse order by this algorithm.

Each of the above algorithms has some theoretical merit but each is inefficient in practice. The first two methods suffer from an excess of superfluous relabelling whilst the third traces each loop in the graph which, although potentially useful, is, as we shall see in § 3.4.3, also potentially very expensive. Johns [49] proposed a method which obviated these problems but an even better solution was given by Tarjan [94]. His DFS method maintains a path and a stack. Each node is added exactly once to both structures and each edge is traversed at most twice.

Thus the time complexity for this algorithm is $O(N + \tau)$ where there are N vertices in the graph and τ edges. The strong components of the digraph are identified by maintaining a pointer for each node which points to the lowest node on the stack to which this node is connected. On backtracking, any node which has its lowlink pointing to itself forms a strong component with all of the other nodes which appear above it in the stack. Duff and Reid [24] have published a Fortran implementation of this algorithm in which they use an improved method of assigning the lowlink pointer. If some node ν_i is the start vertex for an arc which ends on a node ν_j which is below it on the stack, then rather than assigning ν_j to the lowlink of ν_i, the lowlink of ν_j is assigned to this value directly. The same authors [25] compared this code with an implementation of Sargent and Westerberg's [83] algorithm and found the former to perform better in practice. Duff *et al.* [26] have proposed another amendment which improves the performance of this algorithm on undirected graphs. This amendment and other improvements to the algorithm are described in § 5.2.

An entirely different approach is embodied in the preassigned pivot procedure, P^3, developed by Hellerman and Rarick [40]. This is an hierarchical partitioning algorithm which is applied to the whole matrix, whether it is reducible or not, and it requires the concepts of spiked columns, which was introduced in § 3.3, and an active matrix. This is the section of the matrix which contains the rows and columns which are candidates for the next pivotal, *i.e.* diagonal, position. Initially this is the entire matrix, but the active section shrinks at each iteration. In the first step, a search is made for a row, i, which has a single entry in some column, j. Such a row is called a singleton, and this pair is moved to the first position of the permuted matrix, and they are deleted from the active matrix. This is called forward triangularisation and it is repeated on the active matrix

until no more such intersections are located. At this point a similar procedure, backward triangularisation, is performed in which any pair (k, l) such that the entry in column l is in row k is permuted to the last vacant entry in the ordering; again this is repeated until there are no more candidates.

The remaining active matrix is either irreducible or its diagonal blocks are of size greater than unity, and it is to be permuted to bordered block diagonal form. P^3 requires a tally to be maintained of the number of non-zero entries to be found in each row and column. This is necessitated by the desire to produce as many row singletons as possible at each iteration. At each step, if the minimum row count is greater than one, a spike column is transferred from the active matrix to the border. The spike chosen is the column which intersects maximally with the set of rows of minimum row count. In the event of a tie the column with the greatest column count is chosen; if this fails to produce a single candidate the choice is made arbitrarily from amongst the set of columns which satisfy the first two criteria. If the minimum row count is one, and if i is the only row with this count, then then row i and the column with which it intersects are ordered next. If there are $k > 1$ rows with unit row count, and if all of these intersect with the same column, then a diagonal block of size k is formed in the active matrix. The first row of unit row count is paired with the column with which it intersects, and this pairing is ordered first in the new block. The remaining $k - 1$ rows are paired with the last $k - 1$ columns to be identified as spikes, and the complete $k \times k$ block is removed from the active matrix and ordered in the first available position in the new matrix.

Whenever a new pairing has been added to the new matrix the algorithm returns to forward and backward triangularisation, and this process continues until

the entire matrix has been processed. Removing spikes from the border and adding them to diagonal blocks reduces the amount of fill-in experienced during elimination, but it can lead to structurally singular diagonal blocks in matrices which are not themselves structurally singular. Erisman *et al* [29] cite an example due to Westerberg, a private communication, which exhibits this behaviour. This example is shown in figure 3.4 where it can be seen that the 3×3 diagonal block

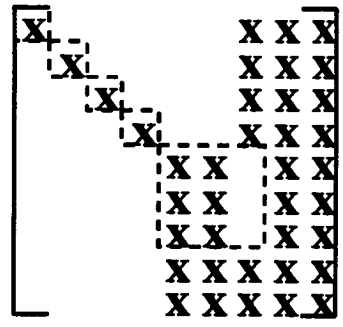

Figure 3.4: Westerberg's P^3 Example

chosen by P^3 is singular; swapping rows seven and eight shows that this matrix is not singular. Erisman *et al* [29] have diagnosed the reason for this, and they have prescribed a modification to the algorithm which corrects this fault. Their algorithm, the precautionary partitioned preassigned pivot procedure, or P^5, is described below.

Effectively P^3 transforms an incidence matrix, A, into one, $\overline{A}$, which is of bordered block diagonal form. By bringing spikes back from the border into the active matrix it produces subblocks along the diagonal of $\overline{A}$, and these may be defined hierarchically. Further, each spike extends at least as far above the leading diagonal as each of the others to its left. This property limits fill-in to those rows in each spike below its first entry. P^3 was used by the same authors to order

the rows and columns within the irreducible blocks found by the partitioned preassigned pivot procedure, P^4 [41]. In this algorithm a maximum transversal is identified and some start node, v_i, is chosen randomly. All paths from this node are traced and the set of all successors of v_i, S_i, is found. This is the set of all nodes which are reachable from v_i. Similarly the set of predecessors of v_i, *i.e.* those nodes from which v_i is reachable, P_i is found. The intersection of these two sets gives C_i, the set of nodes in the same strong component as v_i. The set $\overline{P} = P_i - C_i$ is the set of all nodes which must lie in strong components which precede C_i and $\overline{S} = S_i - C_i$ is the set of all nodes which lie in strong components which follow it. If V is the set of all nodes then $\overline{V} = V - \overline{P} - \overline{S} - C_i$ is the set of all nodes which lie in a disconnected portion of $\mathcal{D}(V, \acute{E})$. The algorithm is repeated recursively on $\overline{P}$, $\overline{S}$ and $\overline{V}$.

Erisman *et al* [29] showed that structurally singular blocks can be produced by P^3 and P^4 because of the way in which spikes are removed from the border and used to form a diagonal block. As an example, when P^3 tries to identify a fifth pivot in figure 3.4, the minimum active row count is two and yet removing a spike reduces three rows to singletons, each of which has its entry in the same column. In this case, only the first two columns of the new diagonal block can be guaranteed pivots, although fill-in may provide the third. This problem arises because the last spike, *i.e.* column 7, was moved to the border when searching for a previous pivot, and hence it was not essential that it contained an entry in any of the rows in the 3×3 block. If the last spike removed from the border had contained a non-zero in either the fifth or sixth rows, but not the seventh, then row swaps within this block would have given a structurally non-singular block without destroying the overall structure imposed by P^3.

In order to obviate this difficulty Erisman *et al* [29] proposed that the size of the diagonal block be bounded above by the minimum row count when the search for a pivot begins. In this case, regardless of the number of row singletons which are produced by the removal of the last spike, each row is guaranteed a pivotal entry in the new block. In fact the new block must be entirely dense. This is because only rows of minimum row count are retained in the search space when a spike has been removed, and so each spike which has been added to the border since the beginning of the search for a new pivot must contain an entry in each singleton produced. Hence the effects of this modification to P^4 are that every diagonal block in the matrix is dense and, because P^5 leaves some spikes in the border which would have been moved forward by P^4, the border of the matrix will be at least as large as that produced by the original algorithm, and possibly larger. The authors show that fill-in must occur in the border to allow a pivot for any row paired with a spike, but which has a zero intersection with it, and hence P^5 provides a stable factorisation of a non-singular matrix.

Lin and Mah [57] showed that structural singularity can be avoided by choosing both a spike row and a spike column. Consider some block, A, in which a row spike, r_s, and a column spike, c_s, have a zero intersection. Let $r_s = [\lambda, 0]$ and ${c_s}^t = [\sigma, 0]^t$. Then elimination on A can be viewed as the matrix product $EA = \tilde{A}$, *i.e.*,

$$\begin{bmatrix} {A_1}^{-1} & 0 \\ -\lambda^t {A_1}^{-1} & 1 \end{bmatrix} \begin{bmatrix} A_1 & \sigma \\ \lambda & 0 \end{bmatrix} = \begin{bmatrix} I & {A_1}^{-1}\sigma \\ 0 & -\lambda^t {A_1}^{-1}\sigma \end{bmatrix} \tag{3.26}$$

If A is structurally nonsingular, and so too is A_1, then the determinants of E and A must be nonzero. Hence the determinant of $\tilde{A}$ must be nonzero and thus so too must be $-\lambda^t {A_1}^{-1}\sigma$. Since A is defined to be structurally non-singular, and a transversal has been identified for A_1, these conditions have been satisfied.

Using this result, the authors extended the ideas in P^3 and P^4 by trying to minimise the size of the diagonal blocks in order to minimise fill-in. If n_i is the number of rows in the i^{th} diagonal block then they define the performance index, P, to be

$$P = \sum_{i=1}^{i=K} {n_i}^2 \tag{3.27}$$

where there are K blocks and at each stage they seek to choose a spike row, r_s, and a spike column, c_s, so as to minimise P. The partitioning problem may then be formulated as the integer programming problem

$$\min_{r_s,\ c_s} P(r_s,\ c_s) \tag{3.28}$$

The authors present an exclusion theorem which greatly reduces the search space for c_s and r_s at each stage. The algorithm starts by finding a maximum transversal and applying P^4 to partition the matrix; each block is placed on a stack. At each stage a block is popped from the stack, a spike column is chosen according to P^3 and the block is forward triangularised. If this partitions the entire block then the last row is the row spike; if not then a row spike is chosen analogously to the column spike and the block is backwards triangularised. The remaining subblock is precedence ordered and $P(r_s,\ c_s)$ is evaluated. This index is minimised by searching for row and column swaps with the present row and column spikes which reduce it. The only candidates for these row swaps are the members of the minimum spanning row set, $\overline{\mathcal{R}}$, which contains all of the non-zeros in the set of active columns which do not intersect with the spike row[4] r_s; the column candidates are defined similarly. When P_{min} has been found precedence ordering continues until the block has been fully reduced.

This algorithm is complicated and computationally expensive. Since, as the

[4] *i.e.* if $\mathcal{R}$ is the set of all rows with entries in these columns then $\overline{\mathcal{R}} \subseteq \mathcal{R}$ is the smallest subset of these rows such that each column covered by $\mathcal{R}$ is also covered by $\overline{\mathcal{R}}$

authors themselves point out [57], the measure of optimality that is used is crude, they present two more simple criteria for spike selection. The first of these restricts the search for column spikes to those which intersect with rows of minimum row count and the second simply accepts the row and column spikes chosen by P^3. All of these algorithms were shown to reduce fill-in and operations count for a problem when compared with P^4. However, these improvements were gained at the cost of a significant increase in the run time for ordering. Stadtherr and Wood [90] reported a further development of the idea of spike selection. They extended Lin and Mah's simplification by ignoring some possible column interchanges and they presented two new algorithms, SPK1 and SPK2. The former is similar to P^3 except that spike selection starts by identifying the row with minimum row count and pivoting in this row on the column which intersects with it, which has minimum column count. All other columns which intersect with this row are stacked as possible spikes, in order of decreasing column count. The matrix is now forward triangularised with more columns added to the spike stack as necessary. Should a zero row count occur at any time then a spike is popped and assigned to it.

The second algorithm is similar to SPK1 but the tie breaking strategy is different. In SPK1 if there is more than one row of minimum row count then the row for which the sum of column counts is maximised is selected since this reduces the degree of the nodes left in the graph by the maximal amount. In SPK2 the row chosen is that for which column deletion leads to the maximum number of minimum row counts. This is more likely to lead to forward triangularisation.

In summary, although P^3 and its variants are very popular, both P^3 and P^4 can lead to zero pivots. Erisman *et al* [29] prescribe a simple solution, P^5, which

requires an amendment to the spike selection algorithm. Lin and Mah [57] show that no zero pivot is possible, in a structurally non-singular matrix, if one chooses both row and column spikes. They use this observation to develop a range of partitioning algorithms, although these are inefficient, and their optimality criteria are poor. Their approach was extended and simplified by Stadtherr and Wood [90] who developed the SPK1 and SPK2 algorithms.

Soylemez and Seider [88] focused on the structural properties of the equations rather than on that of their incidence matrix. They suggested that equations ought to be arranged in order of increasing non-linearity and that symbolic forward substitution method be used to recast the problem. When a set of sufficiently non-linear equations has been identified they suggested that they be solved as a block. Whilst this approach has some intuitive appeal it is of little practical use because it takes no account of the numerical values of the variables and, further, the classification of non-linearity is very crude.

A more sophisticated approach was proposed by Stadtherr *et al.* [89] who introduced the concept of an allowable subset. This is a set of equations which can be solved exactly, *e.g.* a pair of linear or quadratic equations, without resort to iteration. They contended that such equations might occur when the values of some variables became known or assumed (torn) when the equations are precedence ordered. They presented an algorithm which attempts to identify minimal subsets of equations and so check these for 'allowability'. On recognition of such a subset it is permuted to the next vacant entries at the front of the matrix and partitioning continues. Westerberg [101] has warned against this approach (and indeed against hierarchical partitioning in general) since, in his experience, it produces linearly dependent reduced subsets within a significant number of

structurally nonsingular problems.

Perhaps the most widely used permutation algorithm is that due to Markowitz [61]; this is popular with those who solve linear equations. At each iteration a pivot is chosen which satisfies,

$$\begin{aligned} \min_{i,j} \quad & C = (\rho_i - 1)(\gamma_j - 1) \\ s.t. \quad & (i,j) \neq 0 \end{aligned} \tag{3.29}$$

where ρ_i is the number of non-zeros in row i and γ_i is the number of non-zeros in the j^{th} column. This is a strategy of local minimisation of fill-in and C is used rather than $C' = \rho_i \gamma_i$ in order to force the selection of row and column singletons. This method is used as the basis of Duff's MA28 algorithm [25] and it has been shown to be very successful in practice.

3.3.4 Summary

Rose and Bunch [79] showed that partitioning both reducible and irreducible matrices can be advantageous, and Rose [78] demonstrated how fill-in is related to node order in a symmetric graph; Yannakakis [106] proved that finding the minimum amount of fill for any graph is an NP-complete problem. Haskins and Rose [39] attempted to prepare the ground for similar results on digraphs, which so far has proved fruitless, and Rose and Tarjan [80] showed that computing the minimum fill-in for a directed graph is NP-complete. Schreiber [84] demonstrated how fill-in in undirected graphs is determined by the ordering of the nodes, and his results were extended by Liu [58], who proved that fill-in can be characterised by the leaves of an elimination tree.

Harary [38] and Himmleblau [43] both used a symmetric matrix multiplication technique to partition an incidence matrix; both of these methods is algorithmically inefficient. Steward [93] adopted a more efficient approach in which he found a maximum transversal for the matrix, and then ordered it using a depth first search. Even more successful, and considerably more popular, is the minimum degree ordering algorithm due to Walker and Tinney [97] which orders next the node of minimum degree in the signal flowgraph of the incidence matrix.

Sargent and Westerberg [83] were the first authors to present a depth first search method which partitions the rows and columns of a matrix asymmetrically. Both their method and that of Christensen and Rudd [16] are effective but each suffers from a surfeit of relabelling. Forder and Hutchison [32] presented a different approach in which each cycle in the digraph which represents the asymmetric matrix is identified; this search is very expensive. Johns [49] described a much more efficient depth first search, but even better was that due to Tarjan [94]. This algorithm has a time complexity which is linear in the number of nonzero entries in the matrix, which is the lowest possible theoretical bound for this task.

Hellerman and Rarick [40] took an entirely different approach to partitioning a matrix in which they did not attempt to identify its block triangular structure. Rather they tried to minimise the number of 'spikes' in the matrix, columns which had superdiagonal non-zero elements. They extended their ideas, Hellerman and Rarick [41], by applying the same technique to the diagonal blocks of a lower triangularisation of a matrix. Their techniques have enjoyed some success, but they are prone to producing zero pivots. This problem was diagnosed and obviated by Erisman *et al* [29] by a modification to the number of spikes which can be reintroduced to the active matrix when zero rows are identified. Lin

and Mah [57] also offered solutions to this problem, but their algorithms are highly inefficient and it is based upon a questionable optimality criterion, as they themselves indicate. Stadtherr and Wood extended their analysis and produced two algorithms, but neither of these has the theoretical rational of Hellerman and Rarícks' techniques. Yet another, and much simpler, approach was taken by Markowitz who's algorithm minimises the product of the row and column count for the next pivot to be chosen. This technique has been in vogue for a considerable number of years.

3.4 Methods Of Decomposition

Whilst the chemical engineering literature is replete with decomposition methods, it seems that considerably less attention has been paid to this subject in the wider field. The techniques available can be classified into four different groups

1. *Ad hoc* Strategies.
2. Graph reduction methods.
3. Explicit loop breaking techniques.
4. Depth first search methods.

This classification is inexact in that some decomposition algorithms contain elements of more than one approach. In the following discussion both node and

edge tearing strategies will be described, and each of the above groups of methods is dealt with in turn.

3.4.1 Ad hoc Decomposition Methods

Lee, Christensen and Rudd [53] proposed a minimum node tearing strategy based on an exhaustive search. Their argument was based on the observation that the minimum number of tears necessary is bounded below by the minimum *in*-degree of an 'equation' node in a strongly connected digraph. Let this minimum be $\kappa + 1$. In their method all possible combinations of κ tear nodes are tested to see if they decompose the digraph entirely. If they do, success is reported and the search is discontinued. If failure is encountered then an attempt is made to find a tear set of size $\kappa + 1$ and so on until a node separator set for the digraph is determined. In its most basic state this is an expensive algorithm which is prone to combinatorial explosion. It is possible that some improvement on performance might be achieved by ordering candidate tear sets according to the relative success of their ancestors, or by using a branch and bound search method. No such extensions to this technique have been reported.

Himmleblau [43] and [44], presented two separate decomposition algorithms. In the first [43] he proposed tearing the edge between the first vertex $v_i \in V$ in the digraph $\mathcal{D}(V, \acute{E})$ and v_j, the highest ordered node to which it is connected. Following this the nodes in the digraph are reordered and the process is repeated until no more cycles remain. This algorithm has the advantages of simplicity and low complexity but it ignores entirely the structure both of the incidence

matrix for the problem and the equations themselves. In his second algorithm [44] the nodes are grouped according to their degree before an attempt is made to determine a node separator set; as in [53], if the minimum degree of any node is $\kappa + 1$ then the minimum possible number of tears required is κ. The algorithm starts by selecting a node from the set of minimum degree and ordering this first. Nodes of equal degree are appended to the ordered set in turn such that only one new node is connected to that just added. If no such node can be added then another node of higher degree is ordered next if this is connected to only one node not in the ordered set. If no such node is found then a new sequence is started; this introduces at least κ new tears. This process continues until all of the nodes of minimum degree have been ordered whereupon the ordering continues using the new set of nodes of least degree and so on until the entire graph has been ordered. Himmleblau does not indicate how the ordered sequences should themselves be ordered but it would seem appropriate to arrange them in the order in which they were generated. It is difficult to assess the algorithmic complexity of this approach, but the possible requirement for an extensive search for the next node to be added to a sequence implies that it is unlikely to be a low order polynomial.

Liu [59] provided an algorithm which starts with some separator, S, which separates $\mathcal{G}(V, E)$ into two subgraphs, U and V, and then removes nodes from this separator until it is of minimal size. This is achieved by using a flow technique which identifies a subset, $S_0 \subseteq S$, which has an adjacency set in either U or V which is smaller than S_0 itself. He defines an adjacency set in a subgraph, $\mathcal{G}'$, for the set of nodes, S_0, as

$$Adj_{\mathcal{G}'}(S_0) = \{v_i \mid v_i \in Adj(v_j), v_j \in S_0\} \tag{3.30}$$

and he notes that if some separator S separates $\mathcal{G}(V, E)$ into two subgraphs U

and V, and if $Y \subseteq S$, then $\bar{S} = (S - Y) \cup Adj(Y, U)$ separates $U - Adj(Y, U)$ and $V + Y$. If $\mid Adj(Y, U) \mid < \mid Y \mid$ then clearly $\mid \bar{S} \mid < \mid S \mid$. An adjacency set like Y is identified by establishing a maximum matching, M, between S and the larger of U and V, say U. In this case, if there is a set of nodes $\bar{S} \in S$ such that no $\nu \in \bar{S}$ is a terminus of an edge in M, then by definition $\mid Adj(\bar{S}, U) \mid < \mid \bar{S} \mid$, and so $(S - \bar{S}) \cup Adj(\bar{S}, U)$ is a smaller separator for the graph than S. If such a subset is located it is exchanged with S and the process continues. Liu does not provide any complexity measure for this algorithm, but he notes that the minimal separator set is sensitive to the original choice of S. He notes that the minimum degree ordering [97] provides a good starting point, and that in this case most of the computational effort is expended in obtaining this ordering. Although it was developed for use with undirected graphs, Liu's algorithm is equally applicable to those which are directed.

3.4.2 Graph Reduction Methods

Graph reduction methods seek to reduce the search space for tear sets by eliminating some, or all, of the candidates which can never lead to optimality, however this is defined. In general this is achieved by merging or deleting edges of the digraph and it is a technique which enjoys considerable success, particularly when the edges of the graph are weighted. These weights are assigned according to some predefined criteria. For instance, in process simulation, an edge might be assigned a weight which is equivalent to the number of variables associated with the process stream which it represents. In the equation solving context, a weight might describe the desirability of solving an equation for a particular variable.

Christensen and Rudd [16] pointed out that if parallel edges occur between two nodes then either neither or both must be members of the tear set. Based on this observation they proposed that such edges be combined to make one simple edge which, if the edges are weighted, should be assigned a weight equal to the sum of its constituents. Further, they cite the reduction of two-way edges proposed by Sargent and Westerberg [83] which removes such a pair from the digraph and adds one of them to the tear set if the edges are unweighted; should they be weighted then the pair is replaced by a single directed edge which is assigned the difference between their weights and the edge of lower weight is added to the tear set. Christensen and Rudd also introduced the concept of the ineligible edge. Let u and v be two nodes in $\mathcal{D}(V, \acute{E})$ connected by a single edge, e, and let the weight of this edge be ω_{uv}. If the sum of the weights of all of the edges incident upon u is ω_{u+} and those incident from v is ω_{v-} then if either $\omega_{u+} \leq \omega_{uv}$ or $\omega_{v-} \leq \omega_{uv}$ edge e can never be a member of a tear set of minimum weight. This is so because it is always the case that some combination of either the edges incident on u or those incident from v may be torn to the same effect as e but with a lower weight.

Christensen and Rudd [16] used these reductions and the concept of index nodes to find a minimum weight tear set. They defined an index node to be a vertex each of whose incoming or outgoing edges, or both, is eligible. The first step in the algorithm is the reduction of the digraph using the concepts defined above. If the whole digraph is reduced then the minimum weight tear set has been identified. If an irreducible digraph remains then an index node is torn which minimises the increase in the weight of the tear set. The process of edge reduction and node tearing continues recursively until the whole digraph has been reduced. The node tearing strategy which is employed takes into account only the local effect of tearing a vertex, *i.e.* it increases the weight of the tear set by the minimal

amount possible at each iteration, and so this algorithm cannot guarantee to identify a global minimally weighted tear set.

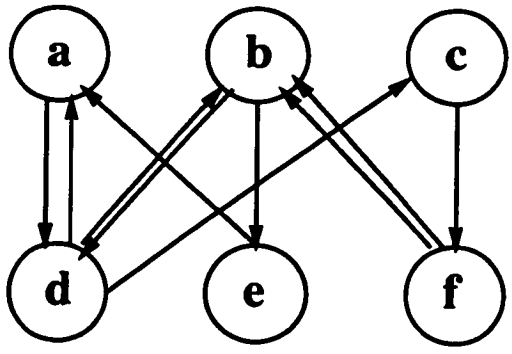

Figure 3.5: An Arbitrary Bipartite Cyclic Graph

As an example of graph reduction, consider figure 3.5, in which there is one pair of parallel edges, and two pairs of two-way edges. Using the graph reduction techniques this is reduced to figure 3.6 and three tears are necessary[5].

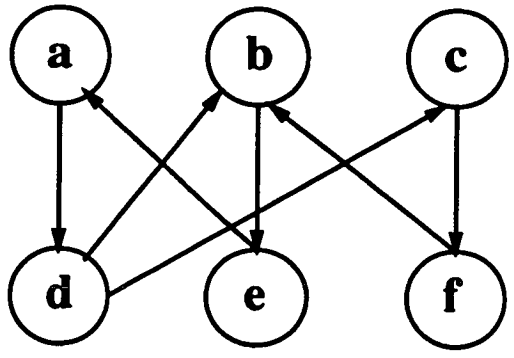

Figure 3.6: A Reduction of figure 3.5

[5] *n.b. either* (a, d) or (d, a) could have been removed from figure 3.5, and similarly either (b, d) or (d, b) could have been added to the tear set

Another graph reduction technique is due to Barkley and Motard [9]. Their algorithm operates on an irreducible signal flowgraph, $\mathcal{H}(V, \tilde{E})$, and so all of the sources and sinks of $\mathcal{D}(V, \acute{E})$ are removed as a first stage. Next $\mathcal{H}$ is reduced to a set of what the authors call intervals. Each of these is a tree, and the set constitutes a spanning forest for $\mathcal{H}(V, \tilde{E})$. These trees are identified by use of the concept of a predecessor. If any node v_i in $\mathcal{H}$ has only one input edge, and that edge is directed from v_j, then v_j is said to be the predecessor of v_i. In this case v_i is deleted from $\mathcal{H}$ and added to the interval which is 'headed' by this node; each edge which was incident from v_i to some other node v_k is removed and replaced by an edge (v_j, v_k) in the reduced signal flowgraph. Should some node v_l be identified as the predecessor of a node v_m which heads some interval, then v_l becomes the new header node for this interval.

This process continues until either all of the nodes in the flowgraph are contained in a single interval, in which case the header node is the only tear variable, or there are no more predecessors in the current subgraph. In the latter case a check is made to see if there are any self loops in $\mathcal{H}$, *i.e.* if any node v_i is a predecessor of itself. Any such loop must be torn since this is the only way in which the digraph can be rendered acyclic. Should any self loops be identified then that which has the highest degree is torn. Ties are broken arbitrarily and the process of interval reduction is restarted.

Two other conditions may be met. If there are no predecessors or self loops in a reduced signal flowgraph then the set of node pairs, $\mathcal{N}$, is found. This is defined on the vertices of $\mathcal{G}(V, E)$ such that

$$\mathcal{N} = \{(v_i, v_j) \mid v_i, v_j \in V, (v_i, v_j), (v_j, v_i) \in E\} \tag{3.31}$$

and exactly one node from each element of $\mathcal{N}$ must be torn since this is the only way in which these minimal cycles can be broken. If $\mathcal{N}$ is non-empty then that node which appears in more pairs than any other is torn, since this minimises the number of tears. Should there be two or more nodes in a maximal number of pairs then the node from this set which has highest degree is torn; if this does not resolve the conflict then a tear node is selected arbitrarily from the set of candidates with the highest degree. If none of the above conditions are encountered then a tear node is chosen either according to degree or arbitrarily. In contrast to Christensen and Rudd's technique [16], Barkley and Motard's method reduces the graph of figure 3.5 to that shown in figure 3.7; it can be seen from this graph that only two edges need be torn to decompose the entire graph. An analysis of the properties and the complexity of this algorithm appear

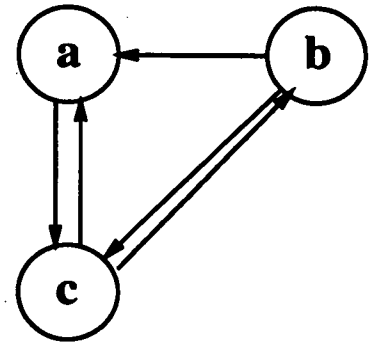

Figure 3.7: The Barkley Motard Reduction of figure

in § 5.2.

Murthy and Hussain [66] proposed a similar approach to that of Barkley and Motard. They assigned a weight to each edge of the digraph and identified the net 'flow' through each node, *i.e.* the difference between the sum of the weights of the edges incident upon the node and the sum of the weights of the edges incident

from it. In their algorithm, any node which had zero or negative flow through it was replaced by its predecessor and the graph was further reduced by using the original predecessor relationship and cutting all self loops. This approach may well tend to minimise the size of the tear set, but there is no guarantee that it will reach or even approximate the global minimum.

3.4.3 Explicit Loop Breaking Strategies

Upadhye and Grens [98] formulated the decomposition problem as the set covering problem so that any tear set chosen would be nonredundant (see § 2.6.1). Taking this approach necessitates the use of a cycle matrix, C, such that $C_{ij} = 1$ if the j^{th} node appears in the i^{th} cycle. If there are N nodes and M cycles in $\mathcal{G}(V, E)$ and if x_j represents the j^{th} node, then the set covering problem can be posed as

$$\begin{aligned} \min_{x_j} \quad & \sum_{j=1}^{j=N} \sum_{i=1}^{i=M} C_{ij} x_j \\ s.t. \quad & \sum_{j=1}^{j=N} C_{ij} x_j \geq 1, \quad i = 1, 2, \cdots, M \\ & x_j = 0 \text{ or } 1, \qquad j = 1, 2, \cdots, N \end{aligned} \tag{3.32}$$

where the first constraint ensures that each cycle is broken at least once. The authors give no advice about the method used to solve this problem, and it is possible that identifying one which is both successful and efficient is problematic. For the set partitioning problem each loop would be broken exactly once and so the $\geq$ condition would be replaced by equality. The partitioning problem is the preferred formulation, but it may have no solution in many cases, and it is likely that it is always difficult to solve.

The first step in any algorithm which solves the decomposition problem in this way is the identification of all of the circuits in a digraph. Tiernan [96] attempted to do this by searching for all of the circuits which exist in each subgraph of $\mathcal{G}(V, E)$. In his method the search starts with the initial vertex in the digraph and a path is constructed through the members of V. If any attempt is made to extend the path by adding the initial vertex, a circuit has been identified and so it is recorded. When all of the edges from a node have been searched it becomes blocked so that no further search is made through this node during the current phase. When a node becomes blocked the search backtracks to the node which was responsible for placing it on the path and the search continues until the initial vertex is removed. At this point the next vertex is used to start the path and all of the other vertices become unblocked. The search continues as before except that any attempt to extend the path by adding a vertex which was used as the start vertex in a previous phase is illegal; this ensures that all of the circuits are traced only once. Whilst this algorithm will identify all of the circuits in a digraph it will examine $\sum_{j=1}^{j=N-1}(N-j)!$ paths for the complete digraph on N vertices.

A more efficient algorithm is that due to Weinblatt [100]. In this case $\mathcal{G}$ is reduced to an irreducible subgraph $\mathcal{D}'$ and each arc of this graph is searched only once. As in Tiernan's method [96], a path is maintained and a depth first search is performed on $\mathcal{D}'$, but in this case a vertex is added to the path once only. Should an arc exist from the vertex at the end of the path P to any already on it then that cycle is recorded. When all of the arcs from a vertex v_i have been searched it is removed from P. Should an earlier vertex v_k on P be connected to v_i then each circuit C_j already found to contain v_i is examined to see if portions of it and any other circuit can be combined to form a new set of circuits C' which contain

v_k or any vertex before it on the path. It is difficult to assess the computational complexity of this algorithm but the examination of previously identified circuits is very costly.

Tarjan [95] presented an algorithm for identifying the elementary cycles of a digraph which is linear in the number of circuits, but which has a worst case time complexity which is exponential in the number of vertices. It uses a depth first search which starts from each vertex in turn and a circuit is detected by an attempt to extend the path by placing the current start vertex on the stack. Like Tiernan's algorithm [96] this circuit avoids retracing circuits by never exploring an arc whose terminus is a vertex numbered lower than that of the initial vertex on the path. Tarjan's algorithm derives its efficiency from the use of a flagging system which avoids searching paths which are known *a priori* to be circuit free. Each time a vertex, v, is added to the stack it is 'marked'. When it is removed, this mark is removed if a circuit has been detected through it; if no such circuit has been found then the node remains marked. If subsequently some node, u, below it in the stack tries to place it back onto the path then this flag is inspected. If v is still marked then no new circuits can be traced through it and so the next member of u's adjacency list is inspected; otherwise a new set of elementary circuits may have been detected.

No linear or low polynomial time algorithm for tracing circuits has been reported.

Having identified the cycles which have to be torn, the next step is to decide on the set of nodes or arcs which must be removed in order to tear them. Lee and Rudd [54] provided an algorithm for this which works by identifying those arcs which must be torn and choosing the others so that either the size or the

weight of the tear set is minimised. Their algorithm is based on the following observations:

1. All self-loops must be broken and the arc added to the tear set. In general these self-loops will occur as a result of a graph reduction and they manifest themselves as a single row entry in the cycle matrix. If column k is the single entry in a some row i, then arc e_k is called an essential arc.

2. If arc e_i appears in every loop in which node e_j appears and if ω_i, the weight of arc e_i is less than or equal to ω_j, the weight of arc e_j, then e_j can never be selected as a tear stream in preference to e_i. Arc e_j is said to be strictly contained within e_i and it is deleted from the loop matrix. This definition can be extended to allow an arc to be contained within a set of others. If the arcs are not weighted then the condition is relaxed so that all that is taken into account is loop membership; this is called containment.

These authors form the loop matrix for $\mathcal{D}(V, \acute{E})$ and they use these rules to reduce it as far as possible. The next step is the formation of the disjoint set of arcs for each cycle, *i.e.* the set of arcs which do not appear in the loop. Clearly, if arc e_i does not appear in cycle C_j then this loop can never be broken by tearing only the i^{th} arc. A tear set of minimum size is the smallest set whose members cannot be generated as a subset of any disjoint set and which covers each cycle in the graph. Lee and Rudd present some rules for restricting the search space for these sets, but their arguments seem to be based on an ability to inspect the initial reduced cycle matrix by eye. They present a version of the algorithm which finds a tear set of minimum weight, but this too requires a large search. In each case a tie breaking strategy based on the number of loops in which an arc occurs is used;

this removes the guarantee that any tear set is of minimal size or weight. Forder and Hutchison [32] used a similar method to Lee and Rudd, but they allowed the user to select some tears *a priori*, based on physical intuition or experience. In this case the optimisation of the tear set is constrained.

Pho and Lapidus [73] used a signal flowgraph to determine the edge tear set of minimum weight for the corresponding digraph. They used the concepts of essential arcs and strict containment described above and they introduced the analogous idea of row containment. Here, if each arc in cycle C_j appears in some other cycle C_k then row k of the cycle matrix can be removed since any tear which breaks C_j must also break cycle k. The first step in the algorithm is the reduction of the cycle matrix as far as possible, using the concepts above and removing self-loops. If this fails to tear all of the circuits then the set of two way edges is inspected (see § 3.4.2). If some arc e_i is involved in two way edges with the arcs in the set $S_N = \{e_1, e_2, \cdots, e_N\}$, then either e_i or each member of S_N must be torn. If the combined weight of the arcs in S_N is less than that of e_i, then each of these is torn; otherwise e_i belongs to the tear set. If some cycles remain unbroken after the reduction of two way edges then a branch and bound method is used to minimise the weight of the remaining tears.

Although it is not a formulation of the set covering problem, Montagna and Iribarren [63] describe a similar iterative procedure which transforms the original digraph into one which is undirected, and then defines a new direction for each of the arcs such that the tear set for the new digraph is of either minimal weight or size. First all of the cycles in the directed graph are identified. Next each arc e_j has associated with it two variables, x_{j1} and x_{j2}, which are used to determine the orientation of this arc in the final graph. At the solution, exactly one of

these variables must be one and the other zero. They formulated this problem in a flowsheeting environment and they interpreted x_{j1} to mean that information flow was in the direction of material flow through the process; in the equation solving environment this would be interpreted as the original relationship holding between an equation and a variable. If the weight of the i^{th} arc is ρ_j, and if p_j is one if e_j is torn, and zero otherwise, the minimisation of the weight of a tear set may be formulated as the integer program

$$\begin{aligned} & \min_{j} \sum_{j=1}^{j=M} \rho_j p_j \\ s.t. \quad & x_{j_1} + x_{j_2} = 1 \quad j = 1, 2, \cdots, M \\ & x_{j_i} = 0 or 1 \qquad i = 1, 2 \end{aligned} \tag{3.33}$$

where there are M arcs in the graph. To find the tear set of minimum size each weight is set to one. This integer program is solved within constraints which arise from the necessity to tear each loop in $\mathcal{D}(V, \acute{E})$ at least once, so that each arc can have a unique direction and each node be correctly connected[6].

The solution to this integer program may direct some of the edges of the new digraph so as to form new cycles and so the program must be reformulated and solved repeatedly until no new circuits are encountered. As evidence of the efficacy of this method, the authors present a new solution to the Cavett [15] problem which has only one torn edge. However, they give no indication of the difficulties involved in solving the integer program. Whereas a global solution to this program is a minimum tear set for the digraph in question, there may be no guarantee that such a solution will be found. Further, no report is given of the difficulty of setting up the constraints in this formulation, and it may well be that this requires a significant amount of work. Thus, although their work is of

[6] *e.g.* a countercurrent heat exchanger must have two inputs and two outputs whereas a mixer has more than one input but only one output.

considerable theoretical interest, it is unknown whether Montagna and Iribarren's [63] approach is likely to be of any practical use.

3.4.4 Depth First Search Decomposition

Motard and Westerberg [65] extended the concept of Upadhye and Grens's [98] decomposition families by defining an exclusive tear set (ETS) to be one which tears each loop in $\mathcal{D}(V, \acute{E})$ exactly once. They proved that if and only if such a tear set exists then the nonredundant decomposition family of Upadhye and Grens is unique, and that each member of it is an ETS. In this case the circuits in the digraph can be ordered as a tree and each ETS can be generated in turn by using the replacement rule round each cycle. The authors presented an algorithm which finds an ETS for a digraph if one exists but, if it does not, it generates a tear set which minimises the maximum number of times that any single circuit is broken; amongst the tear sets of minimum multiplicity that of minimum weight is chosen.

The algorithm operates on the edges and cycles in the digraph. Each edge has assigned to it a weight and an edge efficiency, which is the number of loops which will be broken per unit weight of the edge; this is not necessarily a whole number. The edges are ordered according to their efficiency and weight, those of highest efficiency first and, within a given level of efficiency, those of lowest weight first. A depth first search of these ordered edges is used to find the tear set. At each stage the next edge in the ordering is added to the current tear set until all of the loops have been broken. At this point the weight of the tear set is calculated

along with its multiplicity, the maximum number of times that a cycle has been torn. If the multiplicity of this tear set is lower than the current minimum, or if they are equal but its weight is lower, then this tear set replaces the previous best. Regardless of whether the current tear set is currently optimal or not, the last edge added to it is then removed and the next candidate is added.

In order to prevent forming and checking each possible tear set for optimality, Motard and Westerberg [65] provide an implicit enumeration technique which minimises the search space. If some edge, e_i, cannot be added to a tear set without violating the optimality conditions then this edge is rejected and the next candidate is checked for eligibility. The authors provide an example where this branch and bound technique works well. However, it is not clear that theirs is a practical example and so no conclusions can be drawn as to its practical use. It is clear, however, that complete enumeration would be prohibitively expensive for all but the smallest of problems.

Cordoba [17] has devised a linear time algorithm which identifies a nonredundant tear set for a digraph. It is based on Tarjan's depth first search algorithm [94]. If during the search a back edge from u, the node at the top of the stack, to v, some node below it, is encountered, then this edge must be the last in a cycle which is rooted at v. If the edge to be torn in a cycle is selected always to be the last, then the tear set produced must be nonredundant, although no statement can be made about its minimality. If a forward edge to some node previously on the stack is encountered, then no action is taken. No action is necessary because the only new cycles which can be traced through this edge must be subcycles of those already found and so, since the last edge in each cycle is torn, they must have been broken already.

3.4.5 Summary

The *ad hoc* strategies espoused by the pioneers of decomposition strategies were inefficient in the amount of effort required to identify a solution, and they involved no conditions on optimality. Sargent and Westerberg [83] provided a better approach by introducing the concept of graph reduction. By unifying parallel edges and replacing each circuit of two edges with a single edge, their technique can reduce the search space for a solution considerably. Christensen and Rudd [16] augmented this strategy by finding a minimum weight edge set which spans the cycles in a digraph. A tear edge of minimum weight is chosen from this set and the digraph is reduced; this process continues until no more cycles remain. This approach guarantees a local minimisation of the weight of a tear set, but this does not imply that this is a global minimum. Barkley and Motard [9] also described a graph reduction technique, but this makes use of a spanning forest of the signal flowgraph of the original directed graph. It identifies all cycles in the flowgraph which are of length $l \leq 2$ and tears these accordingly. However, any cycle which is longer than this is torn in an arbitrary fashion. Hence this cannot guarantee that the tear set which it produces is of a minimum size. Murthy and Hussain [66] employed a similar technique on the original digraph, but whilst their rules are simpler and less costly to implement, so too they are less rigorous and there is no guarantee of nonredundancy.

Perhaps the most elegant formulation of the tearing problem is as the set covering or partitioning problem, but this can be difficult to solve. Lee and Rudd [54] developed a similar approach in which a minimum weight cover is found for the cycles in a digraph, and a minimum weight tear set is identified using a combinatorial method. Once again no optimality can be guaranteed because the

technique involves an arbitrary tie breaking strategy. The same criticisms can be made of Forder and Hutchison's [32] algorithm which is closely related to that of Lee and Rudd. Montagna and Iribarren [63] reported another integer programming formulation which is based on the cycle structure of the digraph. The success of their formulation is at the mercy of the solution method used, but it is likely that identifying a tear set will always be an expensive task, and manipulating a problem into the desired form is very difficult.

Motard and Westerberg [65] defined an exclusive tear set to be one which tears each cycle in a digraph exactly once. They provided an algorithm which identifies such a tear set if it exists and, if not, one which minimises the maximum number of times that any cycle is broken. This is a desirable goal, but their technique is based on a combinatorial method which may be prone to explosion. A much more efficient approach is the linear time algorithm devised by Cordoba [17]. This is based on the depth first search method of Tarjan [94] and, whilst it makes no attempt to produce a tear set of minimal size, it will always identify one which is nonredundant.

It has been shown that many of the effective techniques available for decomposing digraphs are inefficient in their exposition, or prone to combinatorial explosion. The only definite condition on optimality for a tear set is that it should be nonredundant, but, from a pragmatic point of view, it is desirable to minimise its cardinality. Using these criteria it appears that the graph reduction method for producing a nonredundant tear set of minimum size due to Barkley and Motard [9] is the best available.

3.5 Conclusions

It has been demonstrated that it is difficult to define any measure of optimality for the selection of an output set for a given problem. Given this, the best method to be used is that which has the best worst case time complexity; this is Dinic's algorithm [20]. No such problem exists for matrix partitioning, however, and the most efficient algorithm is that due to Tarjan [94]. These algorithms have been developed for use in the mathematical modelling software, and they are described in § 4. The provision of optimality criteria for a decomposition strategy is as troublesome as that for an output set, but it is known that nonredundant tear sets are likely to be more efficient than those which are redundant. Cognizant of this it was decided that the best algorithm available for decomposition was that which identified a nonredundant tear set of minimal size and which did so in a relatively efficient way. Barkley and Motard's algorithm [9] best satisfies these criteria, and its use is described in § 5.

If we take in our hand any volume; of divinity or school metaphysics, for instance; let us ask, Does it contain any abstract reasoning concerning quantity or number? No. Does it contain any experimental reasoning, concerning matter of fact and existence? No. Commit it then to the flames: for it can contain nothing but sophistry and illusion.
David Hume, An Enquiry Concerning Human Understanding

Chapter 4

Matching and ordering Variables and Equations

4.1 Introduction

The arguments for finding an assignment for an equation set, and also for partitioning it, were seen in § 2, and methods for achieving these ends were discussed in § 3. It was decided that, in the modelling software, an assignment for the equations would be determined by using a modified version of Dinic's algorithm [20], and that, following this, these equations would be partitioned with Tarjan's depth first search procedure. This order is vindicated in § 4.2, where so too it is shown how these algorithms can be used to find a solvable subset of equations from one which is initially overdetermined, and in particular

how they can be used to replace a redundant equation in a set; the pertinence of these results is demonstrated in § 6.7. A proof of the applicability of maximal flow algorithms to the assignment problem appears in § 4.3 along with a statement of the improved version of Dinic's algorithm. We turn our attention to partitioning in § 4.4, where Tarjan's algorithm is presented and interpreted within the context of equation solving. A short summary of the conclusion drawn from the chapter appears in § 4.5.

Four algorithms are presented in this chapter. Each is stated as a procedure in pseudocode which is a fictitious computer programming language. Only the main operations are shown in order to maximise clarity of presentation. Further, code is shown for only the most important procedures.

4.2 Analysing an Overdetermined Equation Set

4.2.1 The Order of Analysis

Let $\mathcal{G}(V, E)$ be the undirected bipartite graph which describes a square equation set $f(x)$ for which no assignment has been found. It was explained in § 2.4, that the determination of an output set for $f(x)$ transforms $\mathcal{G}(V, E)$ into a directed bipartite graph $\mathcal{D}(V, \acute{E})$, and, as is shown in § 2.5, identifying the strong components of this digraph corresponds to partitioning the equation set into a computational sequence. Theorem 2.2 states that the strong components of this

digraph are independent of the assignment used to form it from $\mathcal{G}(V, E)$; thus the structure which results from partitioning an equation set once an output set has been determined is independent of that output set. Partitioning an equation set before an assignment has been found identifies the components of $\mathcal{G}(V, E)$, which may not be the same as the strong components of $\mathcal{D}(V, \acute{E})$. Hence, if partitioning precedes assignment, a further analysis of the newly directed components of $\mathcal{G}(V, E)$ may be necessary in order to identify its finest grained structure. For this reason, it is better to assign and then partition.

4.2.2 Finding the Minimal Equation Subsets

Any physical problem, Π, has associated with it a set of equations, Ψ, which describe it. These equations may be mass and thermal balances, physical and chemical equilibrium relationships, equations of state, etc. In general there will be more of these equations than are required in order to model Π; some of them may conflict and others may be extraneous. Let the set of all of the variables which appear in these equations be Φ. Then the equation set $\Psi(\Phi)$ may be represented by the undirected bipartite graph $\mathcal{G}'(V', E')$. We will show here how any instance of a generic problem is modelled by some equation set $f(x)$ such that $f \subseteq \Psi$ and $x \subseteq \Phi$, and that $\mathcal{G}(V, E)$, the undirected bipartite graph which describes $f(x)$, is a subgraph of $\mathcal{G}'$. We use this result to show how $f(x)$ can be generated from a general description of the family of problems to which it belongs.

In any instance, $\hat{\Pi}$, of a generic problem Π, some of the variables in Φ will be

known. These may be fundamental or observed constants, or they may be the fixed variables. For instance, if a catalytic reactor is to be modelled then for this problem, the universal gas constant is a fundamental constant, the diffusivity of the bulk gas may be an observed constant, and the yield of one of the products may be a fixed variable. If the set of these knowns, K, is removed from Φ in order to leave the set Θ, the unknowns for the problem, then the equation set which describes it is reduced to $\tilde{\Psi}(\Theta)$.

In general, we are interested only in the values of $\Gamma \subseteq \Theta$, the set of *design variables.* Staying with our reactor problem, it may be that the bulk gas temperature is a member of Γ, but that the temperature of the surface of the catalyst is not. In order to calculate the values of the members of Γ, we need to identify a subset of $\tilde{\Psi}$ which satisfies the necessary conditions for a unique solution which were given in § 2.3. In general, it will not be possible to find a subset of these equations which involve only the members of Γ. Rather, a set of additional variables, Λ will appear in the equation subset too. For instance, in our reactor problem, the specific heat capacity of each gas may have to be calculated in order to determine the reaction temperature. The task of formulating a mathematical model is then the process of identifying a solvable subset of $\tilde{\Psi}$ in which only members of Γ and Λ appear, and such that it embodies no significantly contradictory assumptions.

The problem of contradictory assumptions was addressed briefly in § 1 and it is not considered further here. In order to see how the equation subset is chosen we return to considering $\mathcal{G}'$, the graph of $\Psi(\Phi)$. Each node in this graph which represents a constant or a fixed variable in Φ can be deleted from it, along with each arc of which it is an endpoint, since no equation is required to be solved

for it. Further, we assume that contradictory equations have been removed from $\Psi(\Phi)$ so that, *e.g.*, only one equation of state remains within it. This is a non-trivial task which requires qualitative reasoning. Following this process we are left with a bipartite, undirected graph $\tilde{\mathcal{G}}$ in which the nodes, $\tilde{V}$, are partitioned into $\tilde{V}_v$, the 'variable' nodes, and $\tilde{V}_e$, the 'equation nodes'. If the problem has a solution, then $|\tilde{V}_v| \leq |\tilde{V}_e|$, and it must be possible to devise a maximum matching, $\mathcal{M}$, such that each node which corresponds to some $\nu \in \Gamma$ is an endpoint of one of its edges.

If such a matching is found then $\tilde{\mathcal{G}}$ can be transformed into the directed graph $\tilde{\mathcal{D}}$ in the way described on page 43. We wish to find $\mathcal{D}(V, \acute{E})$, a subgraph of $\tilde{\mathcal{D}}$ which corresponds to $f(x)$, and so we can place the following conditions on it:

1. It must contain an equal number of nodes from $\tilde{V}_v$ and $\tilde{V}_e$.
2. Each subset of κ 'equation' vertices in $\mathcal{D}(V, \acute{E})$ must be adjacent to exactly κ of the 'variable' vertices.
3. For each edge (ν, ω) in $\tilde{\mathcal{D}}$ such that $\omega \in \tilde{V}_e$ and ω is in $\mathcal{D}(V, \acute{E})$, ν is also in $\mathcal{D}(V, \acute{E})$.

These conditions ensure that the strong components of $\mathcal{D}(V, \acute{E})$ represent solvable equation subsets. This follows from theorem 2.1 and the fact that no variable which is not represented in $\mathcal{D}(V, \acute{E})$ can influence the solution of any of the equations which are. There may be many subgraphs of $\tilde{\mathcal{D}}$ which satisfy these conditions, but as a result of the third, each strong component of $\mathcal{D}(V, \acute{E})$ must be a strong component of $\tilde{\mathcal{D}}$, and so we can search for $\mathcal{D}(V, \acute{E})$ by examining the strong components of the larger digraph. We seek a subgraph of $\tilde{\mathcal{D}}$ in which each

strong component, C_i, contains at least one node which corresponds to a design variable, or there is a path from C_i to one which does. This is because there must be a path from each node, ν in $\mathcal{D}(V, \acute{E})$, which represents a member of Λ, to at least one vertex which corresponds to a variable in Γ, *i.e.* the value of at least one design variable is dependent on ν. However, No such path may exist for any node belonging to a strong component of $\tilde{\mathcal{D}}$ which fails to satisfy the connection condition.

It should be noted that, given a maximum matching $\mathcal{M}$ in $\tilde{\mathcal{G}}$, $\mathcal{D}(V, \acute{E})$ is unique. However, if the matching is not complete, the number and membership of the strong components of $\mathcal{D}(V, \acute{E})$ is dependent upon $\mathcal{M}$. As an example of this, consider the undirected bipartite graph shown in figure 4.1. The digraphs which

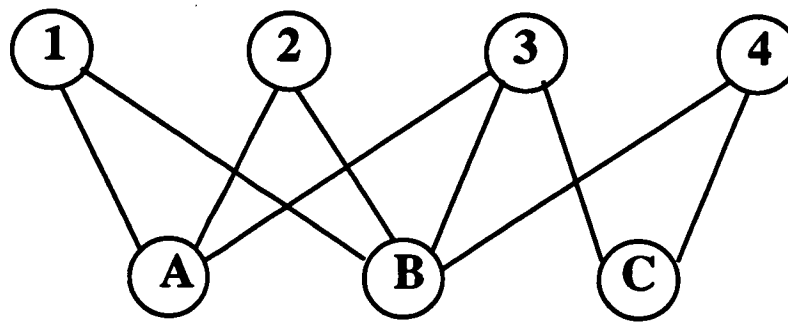

Figure 4.1: An Undirected Bipartite Graph

correspond to two maximum matchings in this graph are shown in figure 4.2. In the first of these there are two strong components, whereas in the second there are three. This observation shows how the existence of a redundant equation, E_r, in a set can be overcome. If the equation set $f(x)$ is formed as described above, and if there is one or more equation which is not involved in a maximum matching, then if the node which represents E_r is removed from $\mathcal{G}(V, E)$, along with all of the arcs incident upon it, then a new matching can be found for $\mathcal{G}(V, E)$, and

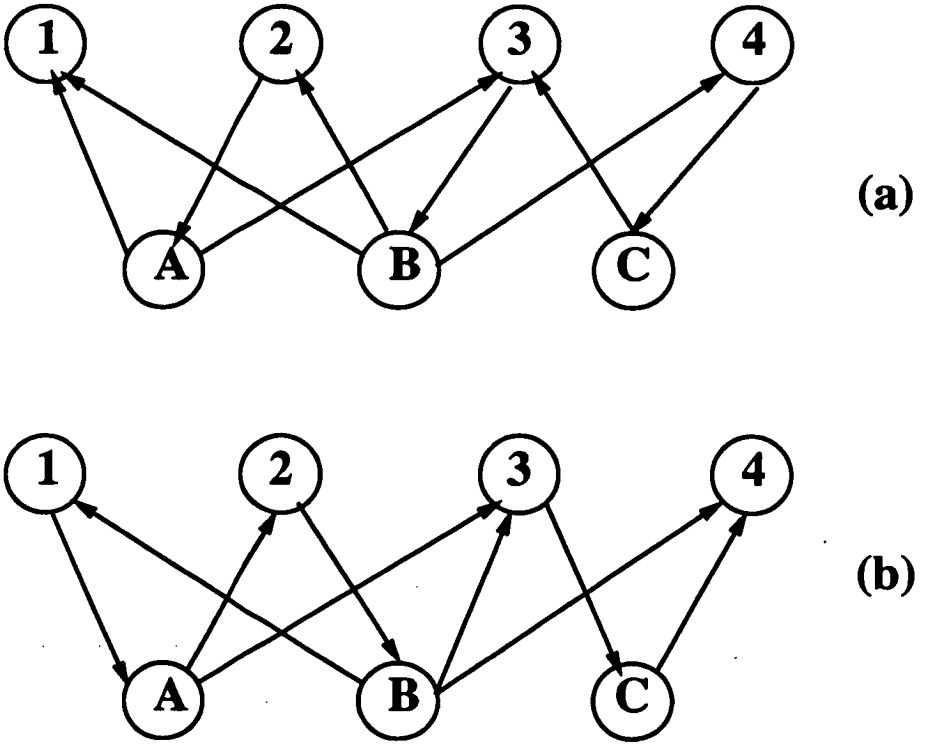

Figure 4.2: Two Directed Versions of figure 4.1

thus a new equation set $\bar{f}(x)$.

Thus we have seen that given a general, overdetermined equation set which is a generic description of a problem, Π, a square subset of it can be identified which can be used to solve a particular instance of Π. In the next section we will discuss how an assignment for the larger equation set can be established, and in § 4.4 we will see how a depth first search can be used to find the required equation subset.

4.3 Finding an Output Set

Recalling the definition of a zero-one network which appears in § 3.2, we use the following two lemmas to show that the assignment problem may be treated as an instance of a maximal flow problem. First we require the well known *max. flow min. cut* theorem, which states that the maximum flow through a network is equivalent to the capacity of its minimum cut. A *cut* for a graph is a set of edges whose removal disconnects it; the set of minimum capacity which satisfies this definition is the minimum cut. Any minimal cardinality edge set, C_{min}, which disconnects a bipartite graph must be a maximum matching for it. This is because each cut for the graph is a matching, but there can be no edge, e_0, in the graph which connects two nodes in different partitions such that neither is the endpoint of an edge in the cut and and $e_0 \notin C_{min}$. Further, no subset of C_{min} disconnects the graph and hence no subset of it can be a maximum matching.

Having shown that assignment in an equation set can be formulated a flow maximisation problem, we describe the algorithm used in the modelling software. It is a modification of an algorithm due to Dinic [20], and it appears as algorithm 4.1. Here $layer(L, Paths)$ constructs a set of augmenting paths through the network. The nodes at the start of these paths are removed from the adjacency set for the sink by $altersinkadj(L, Paths)$ and the matching is updated by $augment(Paths)$.

At each stage in the algorithm a network, $\mathcal{N}_i$, is constructed from $\mathcal{G}(V, E)$ and the current, possibly submaximal, matching $\grave{\mathcal{M}}$ for it. As described above, a source node, s, and a sink node, t are added to $\mathcal{G}(V, E)$. A diedge is added from s to

Algorithm 4.1 The Maximal Flow Algorithm

Procedure max flow

L =Adj(s)

while*(layer(L, Paths) == 1)*
alter sink adj(L, Paths)
augment(Paths)
end

end

□

each $\nu \in \tilde{V}_v$ such that ν is not an endpoint of any edge in $\grave{\mathcal{M}}$; likewise an edge is added from each $\omega \in \tilde{V}_e$ which is not a terminus of an edge in the matching to t. If the maximal flow through $\mathcal{N}_{i-1}$ was $\mathcal{F}_{i-1}$, then $\mathcal{F}_i$ may be greater than this only if flow is channeled through some of these edges. Each useful edge, e, in $\mathcal{N}_i$, *i.e.* one for which the potential flow is unity, is assumed to direct flow in the direction from s to t. The network is then said to be layered, which means that the nodes in the j^{th} layer, L_j, are each *reachable* from s in j edges, *i.e.* there is a path of j useful diedges from s to these vertices.

For each edge (ν,ω) in $\mathcal{G}(V,E)$ such that ν is a variable node and ω is an equation node, then if (ν,ω) is in $\grave{\mathcal{M}}$, so too it must be a directed edge in $\mathcal{N}_i$. If it is not in the current matching then either $\nu \in L_k$ and $\omega \in L_{k+1}$, or this edge is not in the network. This follows from the fact that ω is reachable from s by a shorter path than any which passes through ν.

The zeroth layer in the i^{th} network is $L_0 = \{s\}$, and the edges from this layer are $E_0 = \{(s,\nu)|\nu \in \tilde{V}_v, (\nu,\omega) \notin \mathcal{M}, (\omega,\nu) \notin \mathcal{M}\}$. In $\mathcal{N}_1$, all edges are oriented from

s to t, and thus flow is in this direction alone. This is not true for $\mathcal{N}_i, i > 1$. In this case, L_2, the second layer, is defined as $L_2 = \{\nu|(\omega,\nu) \in E_1\}$ and, immediately following its construction, each vertex, ω, in it must have exactly two edges incident upon it with unit flow. This violates the zero-one condition on the flow through each node, and so one of these flows must be reversed. If the flow through the 'new' edge, $e_1 = (\nu_1,\omega)$, is pushed back, then it can never reach the sink because each node adjacent to ν_1 must have flow channeled through it already. If instead flow is returned along the 'old' edge, $e_2 = (\nu_2,\omega)$, then it may make its way to the sink since no such condition exists on the adjacency set of ν_2. Thus we define the edges in the second layer to be $E_2 = \{(\nu,\zeta)|\nu \in L_2,(\nu,\zeta) \in E,\zeta \notin L_1\}$; *n.b.* each edge in E_2 must have been an edge of E_1 in the previous network.

The definitions of L_2 and E_2, generalised for each layer other than L_0 and E_0, are $L_j = \{\nu|(\omega,\nu) \in E_{j-1}$ and $E_j = \{(\nu,\zeta)|\nu \in L_j,(\nu,\zeta) \in E,\zeta \notin L_k,k < j\}$. The condition $k < j$ is necessary in order to ensure that any augmenting paths found in the network are of minimal length. The process of identifying new layers continues until one of two conditions is met.

If some node $\omega \in L_r$, where r is an odd number, is found such that it is free, *i.e.* flow has never reached this node previously, then the only edge from this vertex must be directed onto the sink, t; a set of augmenting paths has been found. The last layer is identified as the set of all free vertices in L_r, and the layering stops. Note that each augmenting path is of length $l = r + 1$, and that, since the search was breadth first, they must be the shortest such paths in $\mathcal{N}_i$. The other condition is that the edge set for the k^{th} layer is empty, *i.e.* there are no useful edges from the nodes in L_k. In this case no augmenting paths exist in the network, and so the flow must be at a maximum for the original graph; the

search is halted.

In the original algorithm, once the layering has identified a set of free vertices, the next step is a depth first search through these layers for the augmenting paths, starting from each node $\nu \in L_1$ in turn. In our formulation, however, a set of these paths is constructed on backtracking. This requires the use of the set $parents(\nu)$, for each node in the network. This is defined to be the set of nodes, ω, such that $\nu \in Adj(\omega)$ in the current network, and they all lie in the same layer, L_k, as ζ, the node which led to the addition of ν to layer L_{k+1}.

The layering is identified by $Layer(L_0, P_0)$, which is shown as algorithm 4.2. Here L_0 is the set of nodes in the current layer, and the adjacency set for each $\nu \in L_0$ is inspected.

Algorithm 4.2 The Layering Algorithm

Procedure Layer(L_0, P_0)

```
while(pop(ν, L0) == 1)
  blocked(ν) = 1
  copy(Temp, Adj(ν))
  while(pop(ω, Temp) == 1)
    if(ω == t) then
      rest paths(L0, P0)
      push(ν, P0)
      return(1)
    else
      if(blocked(ω) < 1) then
        if(blocked(ω) == 0) then
          push(ω, L1)
          blocked(ω) = -1
        else push(ν, parents(ω))
        end
```

```
      end
    end
  end
end

success = Layer(L1, P1)

if(success == 1) then
  while(pop(Path, P1) == 1)
    phead = head(Path)
    while(pop(u, parents(phead)) == 1)
      if(blocked(u) ≠ 0) then
        push(u, Path)
        push(Path, P0)
        blocked(u) = 0
        pop all(parents(phead))
      end
    end
  end
end

return(success)

end
```

□

If the sink, t, is found in one of these adjacency sets, then the set of nodes $P_0 = \{\nu | \nu \in L_0, t \in Adj(\nu)\}$ is identified by the procedure *rest paths*(L_0, P_0), and the search backtracks. Otherwise, the next layer, L_1, is constructed as described above, and *Layer*, is called recursively. On return, *success* = 0 if flow is a maximum for the original graph, $\mathcal{G}(V, E)$; otherwise it is unity. In the latter case P_1 is the set of potentially augmenting paths of minimum length in the current network. Each of these paths, *Path*, is popped in turn, and its head, *phead*, identified. The set *parents*(*phead*) is examined to see if it contains some node, u, such that no path has been extended through it in the current network. If such a

node is found then its *parents* set is cleared, $Path$ is extended through this node and the the path is added to P_0, the set of potentially augmenting paths which pass through nodes in V_0; should no such node be identified then the path is discarded. This modification has the same worst case time complexity as Dinic's original formulation, but it ought to be more efficient on average.

When the layering procedure has unwound, *alter sink* $adj(L, Paths)$ removes the vertex at the head of each augmenting path from L, the set of free variable nodes, and $augment(Paths)$ changes the matching. This process continues until either L is empty or there are no more augmenting paths, *i.e.* the matching is maximum.

The procedure for augmenting the matching is shown in algorithm 4.3.

Algorithm 4.3 The Matching Augmentation Algorithm

Procedure augment(Paths)

```
while(pop(P, Paths) == 1)
  while(pop pair(A, B, P) == 1)
    if(equation(A) ≠ 1) then
      match(A) = B
      remove(B, Adj(A))
    end
    push(A, Adj(B))
    push(B, P)
  end
end

end
```

□

At first sight this seems to be unnecessarily cumbersome, but this formulation was

necessitated by its implementation in Prolog. This is an object oriented language in which it is not possible to index the elements of a list directly. Although a more elegant formulation of this procedure is possible and expressible in Prolog, *e.g.* using linked lists of structures, their use is less efficient than algorithm 4.3 because Prolog is interpreted rather than compiled, and manipulating its database can be costly. In our implementation, each consecutive pair of nodes on a path is examined. If the first of these, A, is a variable node, then B is matched with it and, since the edge (B, A) must appear in the next network, B is removed from $Adj(A)$. In either case A must be adjacent to B in the next network since either (A, B) is a member of the new matching, or (B, A) is a member of the current one.

As an example of the algorithm in use, consider the initial network for the ideal binary flash equations from appendix B, which is shown in figure 4.3. If the first phase of the matching algorithm were to establish the matching $\{(x_1, 2), (z_2, 3), (V, 4), (6, y_2), (P_t, 7), (P_1, 9), (P_2, 10)\}$, then the second network would be that shown in figure 4.4.

Let each pass through the **while** loop of algorithm 4.1 be called a *phase*; this term was defined originally by Even [30]. The maximum number of matchings possible in $\mathcal{G}'(V', E')$ is $\gamma = min(|V_x'|, |V_y'|) \leq |V'|/2$, and at least one of these must be found in each phase. Thus there may be at most γ phases. Further, each edge in E' is examined at most twice in $Layer(L_0, P_0)$, once when V_1 is being constructed, and then again when P_0 is built. In the worst case only one node is removed from L by *alter sink* $adj(L, Paths)$ and each edge in $Paths$ is searched exactly once by $augment(Paths)$. Hence, each phase of algorithm 4.1 requires at most $O(|E'|)$ operations. Thus the algorithm has an overall worst

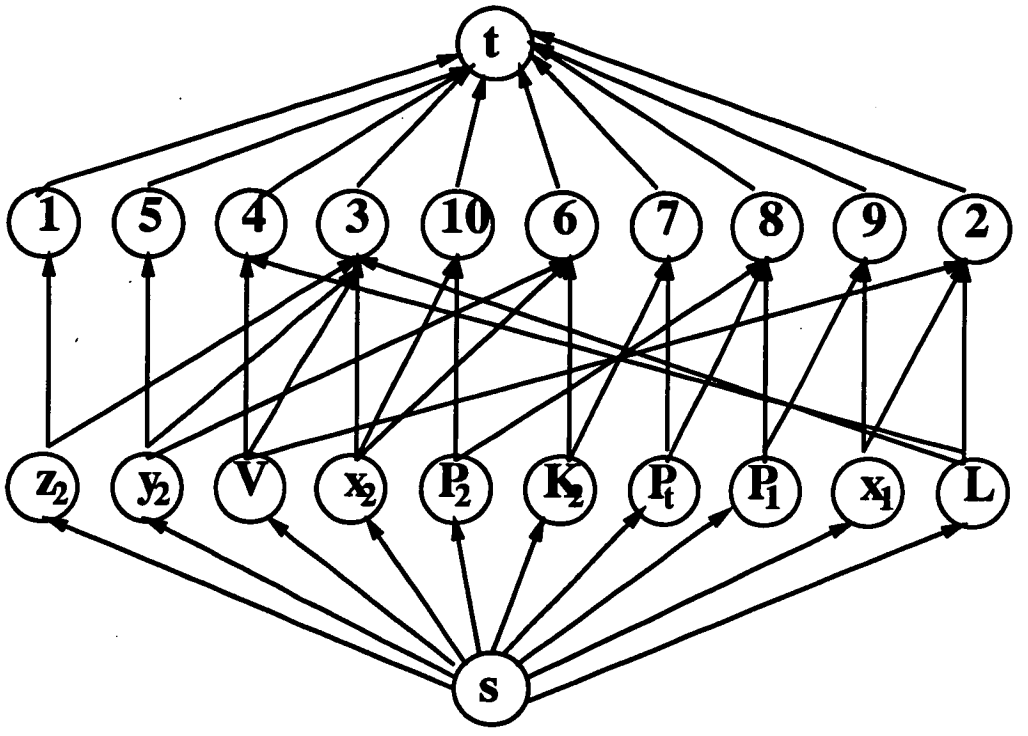

Figure 4.3: The Initial Network for the Flash Equations from Appendix B

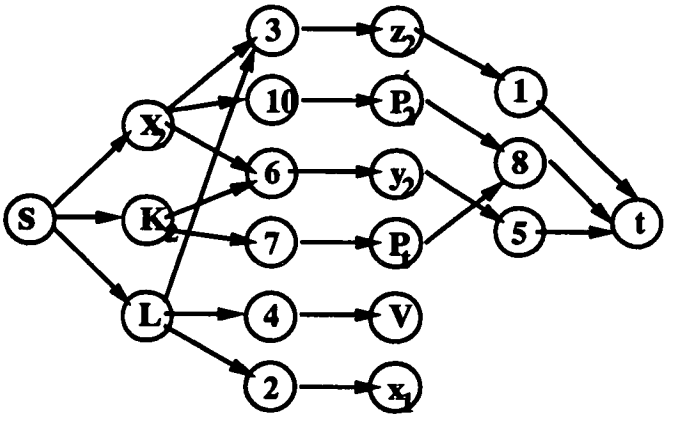

Figure 4.4: The Second Network for the Flash Equations from Appendix B

case complexity of $O(\gamma|E'|)$.

4.4 Selecting and Ordering the Equation Set

The equation set, $f(x)$, is selected from its superset by the procedure *get design*(Γ), which uses the depth first search developed by Tarjan [94]. His algorithm has been adapted by taking into account a suggestion by Duff and Reid [24] which improves its efficiency; a modification which takes into account the bipartite nature of $\mathcal{D}(V, \acute{E})$; and the facts that $|\tilde{V}_v| \leq |\tilde{V}_e|$ and we are searching for a subgraph of $\tilde{\mathcal{D}}$ on $V \subseteq \tilde{V}$. In the algorithm, a stack, $\mathcal{S}$, is maintained which accumulates the strong components of $\mathcal{D}(V, \acute{E})$, and so too is a path $\mathcal{P}$, which directs the search. $\mathcal{P}$ is maintained implicitly since the depth first search procedure is defined recursively.

Two separate orderings are associated with the search, one for the nodes and the other for the strong components. The nodes are ordered as they are added to the stack so that the first to be pushed onto $\mathcal{S}$ is ordered first; the strong components are ordered in the reverse order to which they are popped from $\mathcal{S}$. A lowlink, $\omega = low(\nu)$, is maintained for each node ν during the search. This value is the the lowest ordered node on the stack such that there is a directed path from ν to ω and one from ω to ν, *i.e.* the lowest ordered node which is strongly connected to ν; in this case ω and ν are said to be *reachable* from one another. This is the modification proposed by Duff and Reid [24]. In his original statement of the algorithm, Tarjan [94] defined $\omega = low(\nu)$ to be the lowest ordered node such that there was an arc $(\nu,\omega) \in E$. Using Duff and Reid's suggestion saves some arithmetic operations. A statement of the depth first search procedure is given as algorithm 4.4. Here, $push(\nu, \mathcal{S})$ pushes the node ν onto the stack $\mathcal{S}$, and $pop(\nu, \mathcal{S})$ pops this vertex. If $pop(\nu, \mathcal{S}) = 0$, then no vertex is popped because

the stack is empty.

At each stage, a node, ν, is added both to $\mathcal{P}$ and $\mathcal{S}$, and each node ω in its adjacency set, $Adj(\nu)$, is examined. If $low(\omega) = 0$, then ω has never been examined before and so it is placed on the stack and the path and DFS is called recursively. If, however, $low(\omega) \neq 0$, then either ω is on the stack, or it has been at some point. In either case, $low(\nu)$ is set to the lower of its current value and $u = low(\omega)$. The reason for this is that if u is strongly connected to ω, then there must be a directed path from ν to this vertex; if it is lower on the path than ν, then there must also be a directed path from u to ν, and so they are in the same strong component.

When each member of $Adj(\nu)$ has been inspected, the node is popped from $\mathcal{P}$, and $low(\nu)$ is compared with ν. If $(\alpha = low(\nu)) < \nu$ then these nodes must belong to the same strong component and so ν is left on the stack. If, on the other hand, $low(\nu) = \nu$, then ν is not in a strong component with any node ordered before it on $\mathcal{S}$. Further, each node above it must be in the same strong component as this vertex since it has been left on the stack. Lastly, no node, β, which has been on the stack and popped from it, either before or after ν was pushed onto $\mathcal{S}$, may be a member of this strong component since otherwise ν would have been reachable from these vertices, which clearly it is not. Hence, ν and each of the vertices above it on the stack and the edges between them form a strong component of $\mathcal{D}(V, \acute{E})$. These nodes are popped from $\mathcal{S}$ and stored as a set.

The strong components of $\mathcal{D}(V, \acute{E})$ must be ordered in the reverse order to that

Algorithm 4.4 The Partitioning Algorithm

Procedure DFS(ν, S)

```
  copy(Temp, Adj(ν))

  while(pop(ω, Temp) == 1)
   if(low(ω) ≠ 0) then
     low(ν) = min(low(ν), low(ω))
   else
     low(ω) = ω
     push(ω, S)
     DFS(ω, S)
     low(ν) = min(low(ν), low(ω))
   end
  end
  if(low(ν) == ν) then
   pop stack(ν, S, C)
  end

end
```

□

in which they were popped from $\mathcal{S}$. To see this, consider C_i, a strong component of $\mathcal{D}(V, \acute{E})$ which has just been popped from $\mathcal{S}$. If this was not the only strong component whose vertices were on the stack at that point, then there must be a path from each vertex, γ, on $\mathcal{S}$ to each of the nodes in C_i, and hence C_j, the strong component to which γ belongs, must be ordered before C_i. Further, by construction, there can be no directed path from a vertex in some strong component, C_k, which was popped from $\mathcal{S}$ before it to C_i. Hence either C_i and C_k are disconnected or there is a path from each of the vertices in C_i to each of those in C_k. In either case, ordering C_i before C_k maintains the condition that each edge between strong components in $\mathcal{D}(V, \acute{E})$ is directed from that ordered lower to that which is ordered higher. As was shown by theorem 2.1 this means that the ordering corresponds to a computational sequence for $f(x)$.

In the context of equation solving, when a variable node, ν, is at the head of the path and some equation node, $\omega \in Adj(\nu)$, is pushed onto $\mathcal{S}$, this implies that ν appears in this equation. When this relationship is reversed, *i.e.* the equation node ω is at the head of $\mathcal{P}$ and ν is pushed onto the stack, it implies that ω is to be solved for ν. Hence in the latter case, ω and ν must belong to the same strong component, and thus $low(\nu) \leq \omega$. There are many ways of ensuring that this is the case but, for reasons which will be explained in § 6.7, ω is added to $Adj(\nu)$, even though it is not adjacent to it in $\mathcal{D}(V, \acute{E})$.

Finally we show how the strong components of $\mathcal{D}(V, \acute{E})$ satisfy the conditions set out on page 127. If the matching in $\tilde{\mathcal{G}}$ which is used to form $\tilde{\mathcal{D}}$ is complete, then, by the arguments of theorems 2.1 - 2.3 this must be the case. Consider now the case where $|\tilde{V}_v| \neq |\tilde{V}_e|$, and thus the maximum matching is not complete. No $\nu \in \tilde{V}_v$ which is not matched with a node in $\tilde{V}_e$ can ever be pushed onto $\mathcal{S}$, and hence it can never be identified as belonging to a strong component of $\tilde{\mathcal{G}}$. The same is not true for any excess equation node, however, because any variable which appears in this equation can cause it to be added to $\mathcal{S}$. However, there is no way in which the depth first search can be extended through this node, and so it must be popped immediately. This is identified easily by *pop stack*$(\nu, \mathcal{S}, C)$, since this is the only occasion on which a strong component of only one node can be popped from $\mathcal{S}$. In this case the node is discarded and the depth first search continues.

Having dealt with the depth first search procedure, we can turn our attention to *get design*(Γ), which is shown as algorithm 4.5.

Algorithm 4.5 Identify $\mathcal{D}(V, \acute{E})$ from $\tilde{\mathcal{D}}$

Procedure assign(Γ)

```
while(pop(ν, Γ) == 1)
  push(ν, S)
  DFS(ν, S)
end
```

end

□

At each stage a seed node for the depth first search, ν, is popped from Γ, the set of design variables. This vertex is pushed onto the stack and the depth first search begins. When a strong component is popped from $\mathcal{S}$ by *pop stack*$(\nu, \mathcal{S}, C)$, each $\omega \in \Gamma$ which is popped from the stack is removed from Γ too. Thus, when the stack is empty, either Γ is empty, in which case $\mathcal{D}(V, \acute{E})$ has been identified, or a new depth first search is required. Continuing in this way until Γ is empty concludes the search.

Each node may be pushed onto $\mathcal{S}$ at most once in *get design*(Γ), and each edge in the digraph can be examined once only. Hence the worst case algorithmic complexity for this procedure is $O(|\tilde{V}| + \tau)$, where there are τ edges in $\tilde{\mathcal{D}}$.

4.5 Summary

It was shown in this chapter that a square, solvable subset of equations, $f(x)$, which describes a particular instance of a generic problem, can be identified by

a bidentate strategy of establishing a maximum matching in the graph which describes the equation superset, using this and the original graph to form a new directed graph, $\tilde{\mathcal{D}}$, and then selecting and ordering a subset of its strong components. An algorithm for the matching which operates in $O(\gamma|E'|)$ time, where $\gamma = min(|V_x'|, |V_y'|) \leq \frac{|V'|}{2}$ was presented. This is a modified version of Dinic's [20] maximal flow algorithm. So too a depth first search algorithm which finds a subgraph of $\tilde{\mathcal{D}}$, such that it represents $f(x)$, was presented; this operates with a worst case time complexity of $O(|\tilde{V}| + |\tilde{E}|)$. Lastly, it was shown how those equations not involved in a maximum matching can be used to overcome the incidence of a redundant equation in a simulation.

The great tragedy of Science - the slaying of a beautiful hypothesis by an ugly fact.

T. H. Huxley, Biogenesis and Abiogenesis

Chapter 5

Finding the Minimum Tear Sets

5.1 Introduction

Consider an equation set, $\Phi(X)$, for which an output set, $\mathcal{M}$, has been obtained. Φ may be represented by an undirected, bipartite graph $\mathcal{G}(V, E)$, which is transformed into the directed graph $\mathcal{D}(V, \dot{E})$ by $\mathcal{M}$; this process was described in § 3. $\mathcal{H}(V, \tilde{E})$ is the signal flowgraph for $\mathcal{D}$, defined on the 'variable' vertices. We aim to show that each separator in $\mathcal{H}$ corresponds exactly to one in $\mathcal{D}$, and hence that $\mathcal{S}$, a minimum cardinality non-redundant tear set for $\mathcal{H}$ is also one for $\mathcal{D}$. Further it will be shown that the structure of $\mathcal{H}$ can be represented by the roots of the trees in the spanning forest, $\mathcal{F}$, of $\mathcal{H}$ and the adjacency sets for these trees, and that hence the search for $\mathcal{S}$ may be restricted to a subset of the flowgraph. These proofs appear in § 5.2.

As was indicated in § 3.4, the decomposition algorithm to be used is that due to

Barkley and Motard [9]. In their paper, these authors present only a sketch of this algorithm, and so a fuller statement and description of it appears in § 5.3.2. In § 5.3.1 we show that their algorithm identifies a tear set for a problem, but that an arbitrary tie breaking rule within it casts doubt upon its minimality; this process has an algorithmic complexity of $O(|X|^4)$. Two example decompositions are presented in § 5.4, and, lastly, a summary of the contents of the chapter appears in § 5.5.

5.2 The Signal Flowgraph of a Digraph

Algorithm 5.2 below identifies a tear set of minimum cardinality for the signal flowgraph which is derived from the bipartite digraph which corresponds to a given output set for the problem. In this section we show that this tear set is also a minimally sized separator for the bipartite digraph. So too we demonstrate that each of the nodes which is a candidate for tearing must be the root of a directed tree in the spanning forest[1] for the signal flowgraph, and that a reduction of the signal flowgraph in which only these roots appear can be used to determine the tear set. The membership of these trees is unique but, in general, their ordering is dependent on the matching used to form the signal flowgraph, and so too is the direction of the arcs between them. This effects the cycle structure of the signal flowgraph and hence the membership and minimum cardinality tear set for each output set.

[1]This term is defined in § 2.2.2

Theorem 5.1 *A minimally sized tear set for a signal flowgraph $\mathcal{H}$ is also one for the bipartite digraph $\mathcal{D}(V, \acute{E})$ from which it was formed.*

Proof: Let the vertices in V be partitioned as $V = V_x \cap V_y$, and let $\mathcal{H}$ be formed from the vertex set V_x. Then each path of length l in $\mathcal{D}$ which begins in V_x, and such that l is an even number, corresponds to a path of length $\frac{l}{2}$ in $\mathcal{H}$ which has the same termini. Since each cycle in $\mathcal{D}(V, \acute{E})$ must be of an even length, each of these must correspond to a cycle in $\mathcal{H}$.

Consider one such pair of corresponding cycles, $C \in \mathcal{D}$ and $\tilde{C} \in \mathcal{H}$. Each vertex $\nu_i \in \tilde{C}$ is a member of V_x, and so too it is a member of C; each $\nu_j \in C$ such that ν_j is a member of V_x must appear in $\tilde{C}$. Hence any ν_k which separates $\tilde{C}$ must also separate C. This follows from the fact that each cycle through an equation node must pass through the variable node with which it is matched. Since there is a one to one correspondence between the cycles in the bipartite digraph and its signal flowgraph, any separator for $\mathcal{H}$ must be a separator for $\mathcal{D}$. The converse is true, and so any minimum cardinality tear set for $\mathcal{H}$ must be one for $\mathcal{D}$. □

Having shown that a signal flowgraph can be used to determine the tear sets for the bipartite digraph to which it corresponds, we proceed to demonstrate how the search within it can be restricted to a subset of its vertices. In what follows, each directed tree in $\mathcal{F}$, the spanning forest of the reduced signal flowgraph, is denoted by T_λ, where λ is the root of this tree.

Theorem 5.2 *A subset of the roots of the trees in a spanning forest $\mathcal{F}$ of a signal flowgraph $\mathcal{H}$ yields a minimum tear set for it.*

Proof: Recall that the spanning forest of a signal flowgraph, $\mathcal{H}$, is a set of directed trees such that each node in the flowgraph appears in exactly one of them, and each tree edge is an edge in the flowgraph. The j^{th} tree in this forest comprises of a root node v_j and each node which lies on a path which starts at v_j, and such that it has only one edge incident upon it. An adjacency set can be defined for each of these trees. If V_j is the set of vertices in the j^{th} tree, then its adjacency set, $Adj(T_j)$, is defined as

$$Adj(T_j) = \{u \mid u \in Adj(\nu_i), \nu_i \in V_j\} - (V_j - \{\nu_j\}) \tag{5.1}$$

n.b. if ν_j, the root of T_j, appears in $Adj(T_j)$, then it must not be removed so that self loops can be identified. Any cycle in the flowgraph which passes through some node $\nu_k \in T_j, \nu_k \neq \nu_j$ must also pass through ν_j. Hence each cycle in $\mathcal{H}$ must contain the roots of at least two trees from its spanning forest.

Consider a cycle, C, in $\mathcal{H}$, in which the edge (ν_i, ν_j) appears. Either both ν_i and ν_j are members of the same tree in $\mathcal{F}$ or ν_j must be the root of the j^{th} tree T_j. If the latter condition is satisfied then a new digraph, $\mathcal{R}$, can be produced by replacing the edge (ν_i, ν_j) with (ν_α, ν_j), where $\nu_i \in T_\alpha$, and ν_i is not necessarily distinct from ν_α. Since C must pass through ν_α, there must be a one to one correspondence between the cycles in the two digraphs. If this process is repeated for all directed edges between two trees in the spanning forest in such a way that only one of a set of parallel edges is added to $\mathcal{R}$, and if each subpath, p, in a cycle, such that each node in p belongs to the same tree, T_β, say, is replaced by the root of the tree, ν_β, then each of the cycles in the new digraph must correspond exactly to one of those in $\mathcal{H}$.

Any node which tears a cycle in $\mathcal{R}$ must also tear the corresponding cycle in the original flowgraph; conversely any root node which tears a cycle in $\mathcal{H}$ must tear the corresponding cycle in $\mathcal{R}$. Since each cycle in $\mathcal{H}$ contains at least two root nodes, the search for a minimum tear set for this flowgraph can be restricted to those nodes. □

Note that the signal flowgraph for a bipartite digraph is neither independent of the matching used to form it, and nor is a tear set for one matching is necessarily valid for the other. This is because, in general, the edges between nodes will be directed in different ways and hence the cycle structures of the signal flowgraphs will be different. Consider for example the bipartite graph and two alternative matchings for it which are shown in figure 5.1. The signal flowgraph and its spanning forest for each of these digraphs are shown in figure 5.2. These signal flowgraphs have different cycle structures and, as is shown, they have different minimum cardinality tear sets. That for figure 5.1(a) is $\{B, D\}$, whereas that for figure 5.1(b) is $\{B\}$.

5.2.1 Deriving a Signal Flowgraph from a Bipartite Digraph

Recall that for a bipartite digraph, $\mathcal{G}(V, E)$, with the properties

- $V = V_x \cap V_y$.
- There is a complete matching $\mathcal{M} \in E$ between these vertex sets.

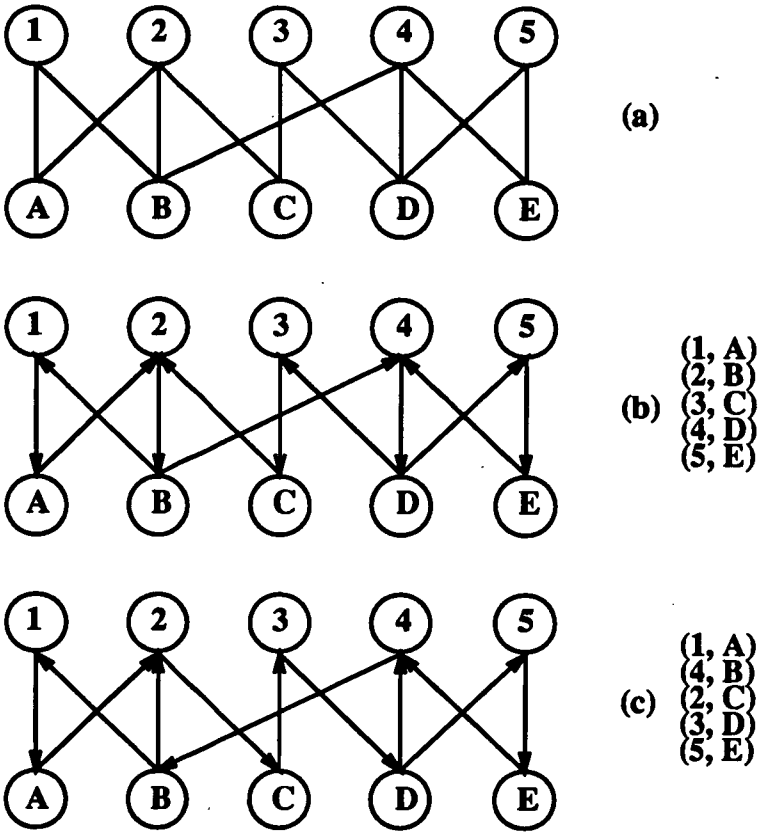

Figure 5.1: A Bipartite Graph and two Alternative Matchings for it

- Each edge $\epsilon \in \mathcal{M}$ is directed from V_y to V_x and each $\epsilon \in E, \epsilon \notin \mathcal{M}$ is oriented in the other direction.

the signal flowgraph which corresponds to it is defined to be $\mathcal{H}(V_x, \tilde{E})$, where

$$\tilde{E} = \{(\nu_i, \nu_j) | \nu_i, \nu_j \in V_x, (\nu_i, \omega_j), (\omega_j, \nu_j) \in E\}$$

The proof that a minimum cardinality tear set for $\mathcal{H}$ is also a tear set for $\mathcal{G}(V, E)$ is given by theorem 5.1. Here we show that this flowgraph can be derived by a breadth first search of V_x which has a time complexity of $O(|E|)$.

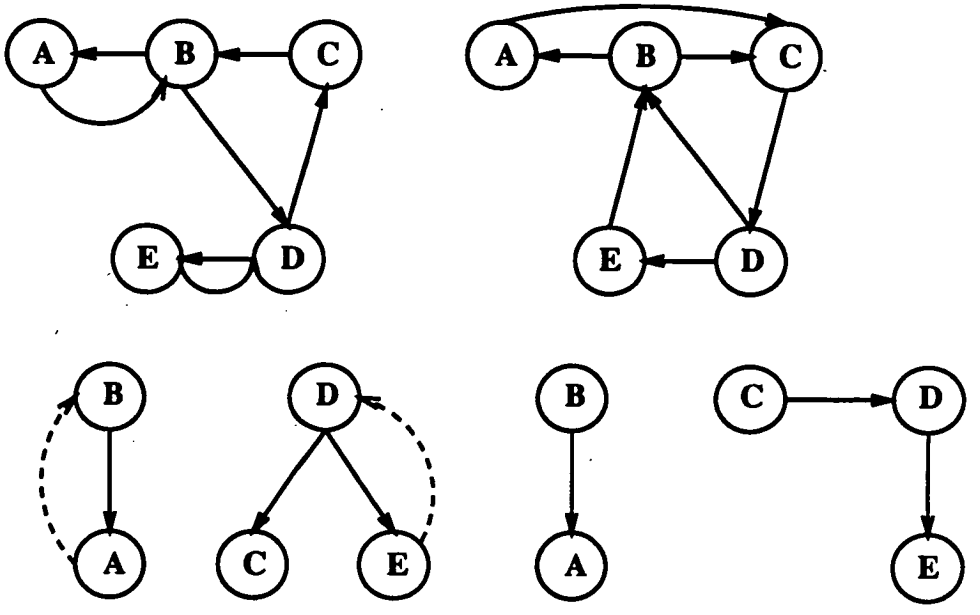

Figure 5.2: The Spanning Forests for the Digraphs of figure 5.1

The breadth first search appears in pseudocode as the procedure *make signal*$(vertices, sigrow, sigcol)$ in algorithm 5.1. Here, *vertices* is an array which holds the members of V_x, and both *sigrow* and *sigcol* are double subscript arrays which contain a version of the reduced signal flowgraph. $sigcol(i)$ is the set of nodes which direct edges onto the i^{th} vertex in V_x; $sigrow(i)$ is the set of vertices which are the endpoints of edges directed from the i^{th} node.

Each node, ν, is popped from *vertices* in turn, and the members of its adjacency set, $Adj(\nu)$, are examined. When ω is popped from $Adj(\nu)$, the node with which it is matched in $\mathcal{D}(V, \acute{E})$, ζ, is added to $sigrow(\nu)$, and ν is pushed onto $sigcol(\zeta)$; *n.b.* only *sigcol* or *sigrow* is required to determine the signal flowgraph, but both

are used for efficiency in the implementation of the decomposition algorithm. Each node in V_x is examined exactly once, as is each edge in E of which it is the initial vertex. Hence both *sigrow* and *sigcol* contain the signal flowgraph which corresponds to the input digraph, and the algorithm has a time complexity of $O(|E|)$.

Algorithm 5.1 Form a Signal Flowgraph

Procedure make signal(*vertices*, *sigrow*, *sigcol*)

```
  while(pop(ν, vertices) ≠ 0)
   copy(Temp, Adj(ν))
   while(pop(ω, Temp) ≠ 0)
    pop(ζ, Adj(ω))
    push(ζ, Temp)
    push(ν, sigcol(ζ))
    push(ζ, sigrow(ν))
   end
  end
end
```

□

5.3 The Decomposition Algorithm

5.3.1 The Rules for Decomposition

Barkley and Motard [9] presented an hierarchy of rules which attempts to find which nodes belong to a minimum cardinality tear set. These are

1. Each node in the flowgraph which has a self loop must be torn.

2. If there is a cycle of length two in the flowgraph, then it is necessary to tear at least one of the nodes in it. If either node in the cycle is involved in more cycles of this form than the other, then tearing it breaks more cycles of length two than if the other were torn. Note that whilst this may lead to a smaller tear set there is no guarantee that it will.

3. If there are no self loops or cycles of length two in an irreducible signal flowgraph, then the node of highest out degree is torn.

4. If this fails to identify a unique next node to tear, then one of the candidates of highest degree is torn arbitrarily.

Rules 3 and 4 can be used to resolve the conflict between a set of nodes, each of which satisfy either of rules 1 or 2.

The justification for the first of these rules is that each self loop may be cut only by tearing the node on which it is defined. Likewise, if a node appears in more than one cycle of length two, then either it must be torn, or each of the other nodes in these cycles must be torn. Hence, tearing the node which appears in more two edge cycles than any other minimises the size of the tear set. If it is necessary to tear both nodes in one of these cycles then the tear set is no longer nonredundant. The last rules attempt a local minimisation of the size of the tear set, but they cannot guarantee its global optimisation.

5.3.2 A Description of the Algorithm

The decomposition algorithm appears as algorithm 5.2. The first steps are the derivation of the signal flowgraph from the bipartite digraph; the identification of the spanning forest of the flowgraph; and production of the reduced signal flowgraph based on the roots of the trees in this forest. At each stage after this a single tear variable, τ, is identified and eliminated from the reduced flowgraph, $\mathcal{H}$, along with each edge directed from or onto it, and this flowgraph is reduced again. The process continues until the reduced flowgraph is empty. At this point a tear set, $\mathcal{S}$, for the bipartite digraph has been identified, and the other vertices have been ordered so that, other than edges directed from torn nodes, all edges in the digraph are directed in the forward direction.

At each pass through the algorithm, even if more than one node which must be torn is identified, only one of these is eliminated from the reduced signal flowgraph. The reason for this is that removing nodes simultaneously may isolate some untorn node, ν, *i.e.* reduce its in-degree to zero, in which case it cannot be assigned membership of any tree in the spanning forest; the position of such a node in the ordering of the torn digraph is indeterminate.

Each tree in, $\mathcal{F}$, the spanning forest of the newly reduced signal flowgraph, $\mathcal{R}$, must be either a tree in the spanning forest for the input reduced flowgraph, $\mathcal{H}$, or it must be comprised of subtrees which were. This is because no new edges are added to $\mathcal{H}$ other than those which replace directed paths, and only edges between trees can be removed; should some tree, T_j, have only one edge incident upon it, (ν_i, ν_j) say, following the elimination of ν, then T_j becomes a subtree of

T_i in the new version of $\mathcal{F}$. The trees in this forest are not stored explicitly, but the vertices within them are stored linearly, subject to a weak ordering, within a link array, *link*. If two vertices, ν_a and ν_b, are both in some tree, T_α, and if there is a path from ν_a to ν_b, then ν_a is ordered before ν_b. The head of each list in *link* is either a node in $\mathcal{R}$ or it is a tear variable. If a node, ν_β, is to be added to a tree T_α[2], then the entry for ν_α in *link* is inspected. If $link(\alpha) = 0$, then this entry is set to β. Otherwise, if $link(\alpha) = \gamma$, then $link(\gamma)$ is examined, and so on, until some entry $link(\delta)$ is found to be zero; then $link(\delta)$ is set to β.

Algorithm 5.2 The Tearing Algorithm

Procedure tear signal(*vertices*)

```
  make signal(vertices, sigrow, sigcol)
  reduce graph(vertices, link)

  while(not(empty(vertices))
   if(self loop(sigrow, τ) == 0) then
    if(double edge(sigrow, τ) == 0) then
     tear largest(sigcol, τ)
    end
   end
   remove tear(vertices, τ)
   push(τ, S)
   reduce graph(vertices, link)
  end
  print tears(S, link)
end
```

□

In *tear signal*(vertices), *vertices* is the list of nodes in the reduced signal flowgraph, $sigrow(i)$ is the adjacency set for ν_i in this digraph, and $sigcol(i)$ is the set of vertices which direct edges onto this node. The procedure

[2] *n.b.* ν_β may be the root of a tree T_β in which case the whole of T_β becomes a branch of T_α.

self loop$(vertices, \tau)$ searches for self loops on the vertices of $\mathcal{H}$, and *double edge*$(vertices, \tau)$ looks for cycles of only two edges. If both of these fail, then *tear largest*$(vertices, \tau)$ tears one of the nodes of maximum out-degree. In each case, τ is the tear node identified. This vertex is eliminated from $\mathcal{H}$ by the procedure *remove tear*$(vertices, \tau)$, and it is pushed onto $\mathcal{S}$, the stack of torn nodes. The flowgraph is reduced to $\mathcal{R}$ by *reduce graph*$(vertices, link)$, which updates *link* as necessary. Finally, when a tear set for the flowgraph has been identified, *print tears*$(\mathcal{S}, link)$ prints out the vertices in their new order.

Searching for self loops is a simple computational task, but the procedure used has a worst case complexity of $O(|V_x|^2)$. Determining the node of highest out-degree is easy too, and it has a worst case complexity of $O(|V_x|)$.

Identifying cycles of length two requires a great deal of effort, and an algorithm for this appears as algorithm 5.3. The adjacency set for each node, ν, in the flowgraph is examined in turn. If the adjacency set for any node $\omega \in sigrow(\nu)$ contains ν, then one of these cycles has been identified. The number of these cycles in which ν appears is calculated and it is compared with the highest number found so far. If ν appears in the same number of cycles as the previous maximum, then this node is added to *dbls*, the stack of variables which appear in the largest number of two edge cycles. If it appears in more cycles than this, then it becomes the only candidate for tearing.

In the worst case, the first **while** loop may be executed $|V_x|$ times, and the adjacency set for each vertex may contain all of the other $|V_x| - 1$ nodes. Thus *double edge*$(vertices, \nu)$ has a worst case time complexity of $O(|V_x|^3)$.

Algorithm 5.3 Find two way edges

Procedure double edge(*vertices*, τ)

```
max pairs = 0
while(pop(ν, vertices) ≠ 0)
 while(pop(ω, sigrow(ν)) ≠ 0)
  if(member(ν, sigrow(ω)) then
   pairs = pairs+1
  end
 end
 if(pairs == max pairs) then push(ν, dbls)
 else
  if(pairs > max pairs) then
   max pairs = pairs
   popall(dbls)
   push(ν, dbls)
  end
 end
end
if(sizeof(dbls) > 0) then
 if(sizeof(dbls) == 1) then
  pop(τ, dbls)
 else
  find max adj(τ, dbls)
 end
end
```

□

The spanning forest for the reduced signal flowgraph is found by *reduce graph*(*vertices*). Each node in the flowgraph, ν, is inspected in turn. If it has only one edge incident upon it, and this is directed from ω, then ν must belong to the same tree in the spanning forest of $\mathcal{H}$ as ω; indeed, by construction, ω must be the root of this tree, T_ω. Each node which is the endpoint of an edge directed from ν is added to the adjacency set for ω; ν is appended to the list of nodes in T_ω and then eliminated from the flowgraph.

Algorithm 5.4 Reduce the Graph

Procedure reduce graph(*vertices*, *link*)

```
  while(pop(ν, vertices) ≠ 0)
   if((ω = single entry(sigcol(ν))) > 0) then
     merge rows(sigrow(ν), sigrow(ω))
     replace row index(sigrow(ν), ω, sigcol)
     add link(ν, ω, link)
     reduce graph(vertices, link)
   else
     reduce graph(vertices, link)
     push(ν, vertices)
   end
  end
end
```

□

The algorithm for this appears as algorithm 5.4.

Merging the two adjacency sets in *merge rows*($sigrow(\nu), sigrow(\omega)$) requires at most $O(|V_x|)$ operations, but replacing ν with ω in each array in *sigcol* may take $O(|V_x|^2)$ operations; adding ν to T_ω has a worst case complexity of $O(|V_x|)$. Hence, *merge rows*($sigrow(\nu), sigrow(\omega)$) has worst case time complexity of $O(|V_x|^2)$.

Each of the main tasks in the algorithm has been examined, and the most expensive of these, for moderate or large $|V_x|$, is *double edge*($vertices, \nu$), which has a worst case time complexity of $O(|V_x|)^3$. Since this may be called $|V_x| - 1$ times, the overall worst case time complexity for the decomposition algorithm is $O(|V_x|)^4$.

5.4 Two Examples

Consider the binary ideal flash problem described in appendix B. The graph of this equation set is shown in figure 2.5, and the digraph for an assignment of these equations appears in figure 2.6. The signal flowgraph which corresponds to the irreducible section of this digraph is displayed in figure 5.3, and its spanning forest

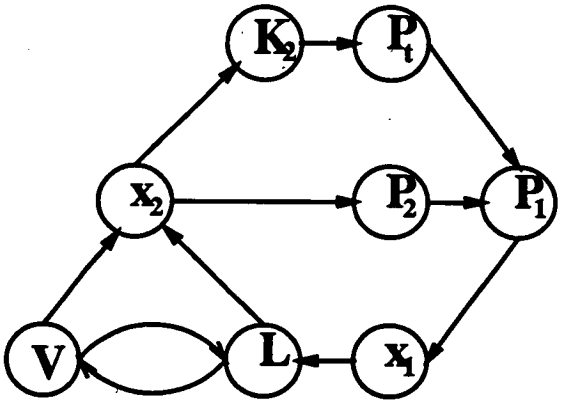

Figure 5.3: Part of the Signal Flowgraph for the Ideal Flash Equations

is shown in figure 5.4, where the broken arcs indicate back edges within trees, and cross edges between them. The structure of this spanning forest indicates that the signal flowgraph can be reduced to the digraph shown in figure 5.5. Here we see that there is a self loop on node L, and so this must be torn. Eliminating this node breaks the two edge cycle between L and x_2, and the digraph has been rendered acyclic. The correct ordering of the nodes is then $\{L, V, x_2, K_2, P_T, P_2, P_1, x_1\}$.

A more difficult example is the 6×6 equation set for which the digraph for the output set $\{(1, A), (2, B), (3, C), (4, D), (5, E), (6, F)\}$ is shown in figure 5.6. The signal flowgraph for this equation set based on the 'variable' nodes is shown in figure 5.7, and its spanning forest appears in figure 5.8. The decomposition

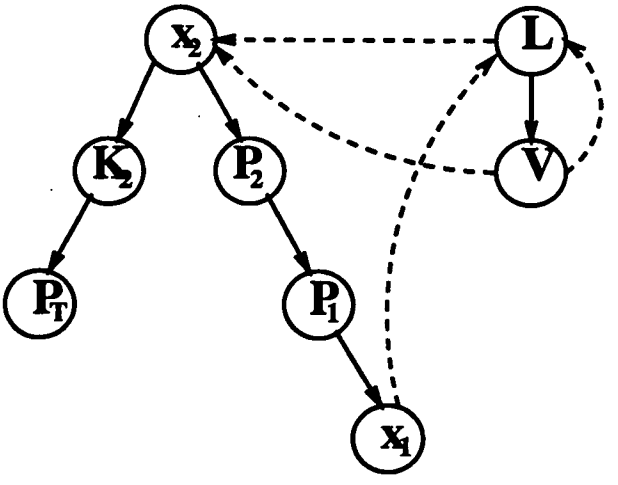

Figure 5.4: The Spanning Forest for the above Signal Flowgraph

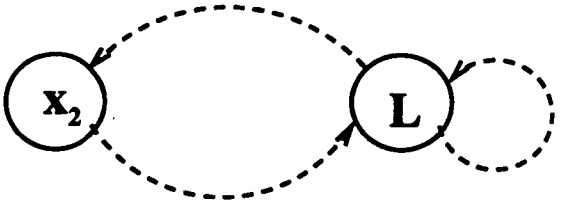

Figure 5.5: The Reduced Signal Flowgraph

algorithm would reduce figure 5.7 to the digraph shown in figure 5.9, where there are self loops on both of its vertices. The algorithm would tear one of these and then the other, so that the minimum cardinality tear set for the problem is $\mathcal{S} = \{B, E\}$. Note that this tears one of the cycles in figure 5.6 twice. One of the minimum cardinality tear sets which avoids a double tear is $\mathcal{S} = \{B, D, F\}$.

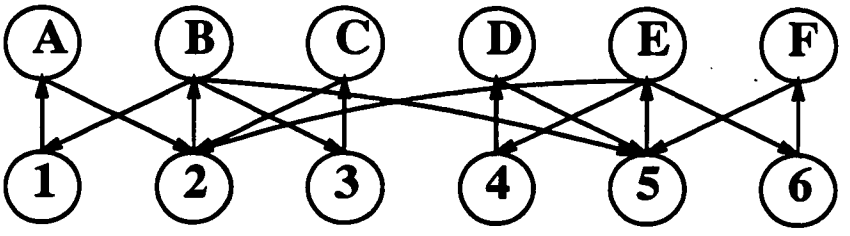

Figure 5.6: A Digraph for a 6×6 Equation Set

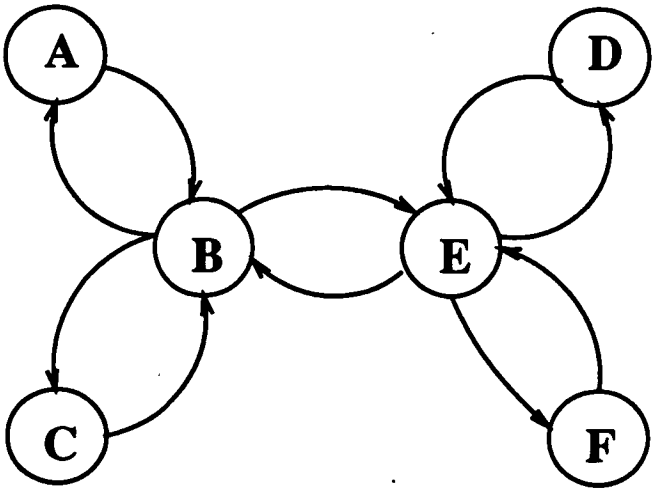

Figure 5.7: A Signal Flowgraph for figure5.6

5.5 Summary

In this chapter we have seen that a minimum cardinality tear set, $\mathcal{S}$, for a reduced signal flowgraph, $\mathcal{H}$, is also a separator of minimal size for the bipartite digraph, $\mathcal{D}$, from which it was derived, and that the search for the members of this set can be restricted to the roots of the directed trees in $\mathcal{F}$, the spanning forest of $\mathcal{H}$. Further, it was demonstrated that the size and membership of this tear set is not necessarily unique for the undirected bipartite graph, $\mathcal{G}$ which underlies

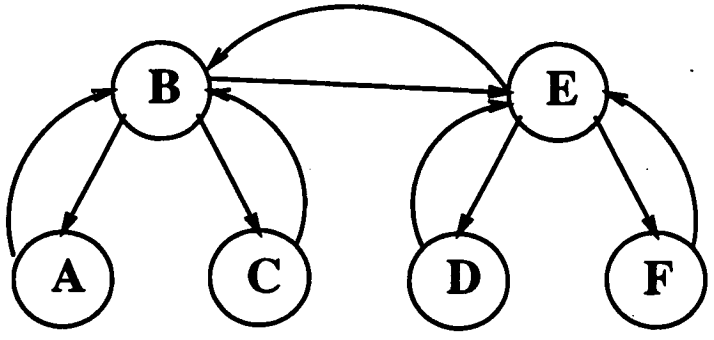

Figure 5.8: The Spanning Forest of the Signal Flowgraph

Figure 5.9: The Reduced Signal Flowgraph

$\mathcal{D}$. In § 5.2.1 an algorithm was presented which will derive $\mathcal{H}$ from $\mathcal{D}$ in $O(|E|)$ operations, where there are $|E|$ arcs in the bipartite digraph. The rules for the decomposition procedure were presented in § 5.3, and algorithms for its more important sections were stated. Lastly, it was shown that this process has an algorithmic complexity of $O(|V_x|)^4$, where V_x is the vertex set in $\mathcal{H}$.

O! Thou hast damnable iteration, and art, indeed, able to corrupt a saint

William Shakespeare, Henry IV, Part 1

Chapter 6

The Generation of Analytical Derivatives and their use in an Equation Solver

6.1 Introduction

There are many techniques available for accelerating the convergence of non-linear equations, and some of them are described in appendix D. It was decided that the Newton Raphson method would be used to solve the equations generated by the mathematical modelling software, and that the Jacobian would be generated analytically. These derivatives are calculated simultaneously with the functions, using a data management technique for torn systems which was developed from one due to Ponton [75] and which is described in § 6.3. This method avoids differentiating and flattening expressions, and so too it minimises the number of times which transcendental functions must be calculated, but these advantages are tempered by the fact that the equations are interpreted, rather than evaluated in the normal way. So too this method fails to address the surfeit of calculation

associated with evaluating derivatives which can be seen, *a priori*, to be zero everywhere. The rules for an algorithm which obviates this difficulty are outlined in § 6.6.

6.2 The Newton Raphson Method

Consider the solution of $f : \mathbb{R}^N \rightarrow \mathbb{R}^N$, a set of N non-linear equations in N variables. At each iteration of the Newton Raphson method, these equations are approximated by a set of linear equations which have the same values as f at the current point x, and identical derivatives. The solution of this approximate model, x^*, is given by

$$x^* = x + J^{-1}f \tag{6.1}$$

where J is the Jacobian for the system. This process is repeated until $f \approx 0$.

If some of the variables in the system have been torn, then a different computational scheme is required. Let the c torn variables be $\hat{x}$ and the others be $\check{x}$, so that $x^t = [\check{x}, \hat{x}]^t$. Each of the first $N - c$ equations are rearranged to give an explicit expression for one of the members of $\check{x}$. Let these equations be $\check{f}$. The remaining c equations, the *kernal* equations $\hat{f}$, are to be solved simultaneously as a set of reduced equations, *i.e.* the solution of $\hat{f}(\check{x}(\hat{x}), \hat{x}) = 0$ is sought. If $\nabla_{\hat{x}}\check{x}$ is the rate of change of the dependent variables with respect to the independent variables, if $\nabla_{\check{x}}\hat{f}$ is the rate of change of the kernal equations with respect to the dependent variables, and if $\nabla_{\hat{x}}\hat{f}$ is the rate of change of these equations with respect to the independent variables, then the computational scheme for this is

1. Guess $\hat{x}$.
2. Calculate $\check{x}(\hat{x})$ and $\nabla_{\hat{x}}\check{x}$.
3. Find $\hat{f}(\check{x}(\hat{x}), \hat{x})$: if $\mid \hat{f} \mid \approx 0$ then stop.
4. Evaluate $J = \nabla_{\hat{x}}\hat{f} \times \nabla_{\hat{x}}\check{x} + \nabla_{\hat{x}}\hat{f}$
5. Set $\hat{x}_{new} = \hat{x} - J^{-1}\hat{f}$.

6.3 The Generation of Analytical Derivatives

It is a simple task to write down the rules for differentiating equations in infix form, and to code these as a computer program. In practice, however, this can lead to the generation of an excessive number of terms in a derivative. Consider, for example, the equation

$$f = 3x - \frac{x^2}{2x - a} + \frac{b - c}{ax} \tag{6.2}$$

Differentiating this using the usual rules would generate the expression

$$\frac{df}{dx} = 3 - \frac{(2x - a)2x - x^2 2}{(2x - a)^2} + \frac{ax0 - (b - c)a}{(ax)^2} \tag{6.3}$$

in which there is one zero term and several which can be combined to reduce the size of the equation. Removing this zero term and modifying the others is called *flattening*, and it is not necessarily unique. For example, equation 6.3 can be flattened to give either

$$\frac{df}{dx} = 3 - \frac{2x}{2x - a}(1 - \frac{x}{2x - a}) - \frac{b - c}{ax^2} \tag{6.4}$$

or

$$\frac{df}{dx} = 3 - \frac{2x(x-a)}{(2x-a)^2} - \frac{b-c}{ax^2} \tag{6.5}$$

Generating either of these forms of the derivative of equation 6.2 requires the comparison of many terms within equation 6.3, and this can be an operose process, even a for relatively simple equation such as that above.

Ponton [75] has shown that these problems may be obviated if the equations are stored and evaluated in Reverse Polish Notation (RPN). This is a form of postfix notation in which an equation is represented as a string of symbols, each operator appearing after its operands. The RPN representation of an infix equation can be generated by parsing it with the rules

Term	*ParsedForm*
Left = Right	*Right Left =*
Left Op Right	*Left Right Op*
Left Op	*Left Op*
Op Left	*Left $\tilde{Op}$*

where the second refers to binary operators, the third postfix unary operators and the last to prefix operators of unit arity; here $\tilde{Op}$ is the reverse form of Op, *e.g.* the $-$ in a term $-a$ would become the reverse subtraction operator $\leftarrow$. Using these rules, equation 6.2 becomes the string

$$3x\times x2\uparrow 2x\times a - / - bc - ax\times / + f = \tag{6.6}$$

In order to evaluate a RPN string, S, it is manipulated in conjunction with a calculation stack, C. Each symbol is popped from S in turn. If it is a variable or a constant, its value is pushed onto C; if it is an operator, the relevant number of elements are popped from the calculation stack, they are operated on, and

the result is pushed back onto C. This process continues until the next symbol popped from S is the variable whose value is sought. At this point the only element in S is this value and so it is popped and assigned accordingly.

Ponton [75] showed that for a single equation in only one unknown, the value of the analytical derivative of this equation can be calculated simultaneously with that of the equation itself. This requires that a derivative stack, D, be maintained and manipulated in the same way as the function stack. As an operand is pushed onto the function stack, so too the value of its derivative is pushed onto D. If the operand is a constant, then this value must be zero; if it is the variable in the equation, then it must be one. When values are popped from C and operated on, the corresponding elements of D are manipulated, possibly with those from C, according to the usual rules of differentiation. For instance, if ν and ω are the top two elements of C, and if ν_d and ω_d are the top two elements of D, then if the multiplication operator, $\times$, is popped from S, ν and ω are popped from C and the value of $\nu \times \omega$ is pushed onto this stack; simultaneously ν_d and ω_d are popped from D and replaced by $\nu_d \times \omega + \omega_d \times \nu$.

Whilst this approach is wasteful in that it may involve a number of additions or multiplications involving zero, it is beneficial in that it is no longer necessary to differentiate the equations explicitly and hence no expression flattening is required. Further it can save effort in calculating expensive terms such as transcendental functions. For instance, the derivative of

$$f = e^{x^2} - e^x \tag{6.7}$$

is

$$\frac{df}{dx} = 2xe^{x^2} - e^x \tag{6.8}$$

Evaluating equation 6.7 and its derivative in the traditional way requires four calculations of the exponential function, whereas in Ponton's method, only two are necessary.

6.4 Application to Torn Systems

As Ponton noted, this technique can be extended to allow the differentiation of $f(x) : \mathbb{R}^N \rightarrow \mathbb{R}^N$ with respect to each of its variables by manipulating a derivative stack for each variable. This was extended to account for the occurence of both dependent and independent variables in a problem. Here the three sets of derivatives described in § 6.2 are required, but one set of derivative stacks, that for the dependent variables, suffices. The values of $\nabla_{\check{x}}\check{x}$ are calculated by pushing a 1 onto the relevant derivative stack each time a tear variable is pushed onto the calculation stack, and a 0 otherwise; when the value of a dependent variable is popped from C, its derivative with respect to each of the tear variables is popped from the derivative stacks. The values of $\nabla_{\check{x}}\hat{f}$ are calculated as before. No explicit calculation is required for $\nabla_{\check{x}}\hat{f}$, however, since all that is required is that each time the value of some variable $x_i \in \check{x}$ is added to the calculation stack, its derivative with respect to each tear variable x_j, $\frac{dx_i}{dx_j}$, is pushed onto the correct derivative stack.

6.5 An Example Problem

Consider the solution of the 3×3 equation set

$$\begin{aligned} x_1 \quad\quad - \quad 2x_3 &= 1 \\ x_1 - 3x_2 \quad\quad &= 2 \\ x_2 + 3x_3 &= 3 \end{aligned} \tag{6.9}$$

In accordance with the computational scheme described in § 6.2, this is solved as

$$\begin{aligned} &1\ 2\ x_3 \times + x_1 = \\ &x_1 2 - 3/x_2 = \\ &\\ &x_2 3 - 3/x_3 - f_1 = \end{aligned} \tag{6.10}$$

where the value of x_3 has been torn.

Let the guess for x_3 be 1, and let S be the function stack, and D be the derivative stack. The calculation of x_1 is shown in figure 6.1. Firstly 1 is pushed onto S and, since this is a constant, 0 is pushed onto D. 2 and its derivative are pushed

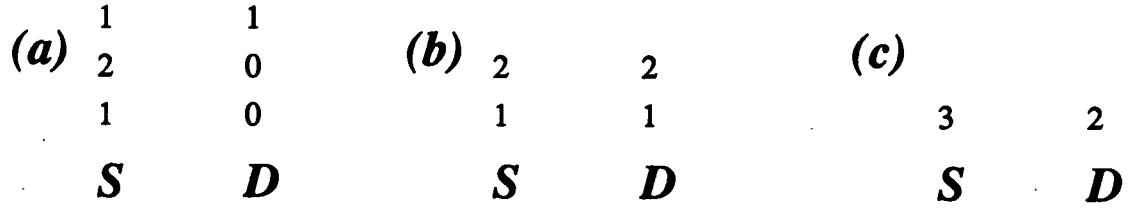

Figure 6.1: The Calculation of x_1

onto the stacks, and then this is repeated for the value of x_3. The next element in the input string is the multiplication operator, and hence the top two elements in

each stack are popped and combined[1]. Since the addition operator appears next on the input, the two elements of each stack are popped and summed, and the result is pushed back; there are no more arithmetic operations and so the value of x_1 is popped from S, and that of $\frac{dx_1}{dx_3}$ is popped from D.

The calculation of x_2 proceeds in a similar manner, as is demonstrated in figure 6.2, and so too does that of f_1; this calculation is shown in figure 6.3.

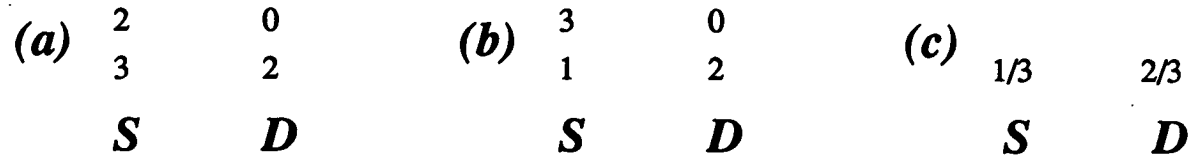

Figure 6.2: The Calculation of x_1

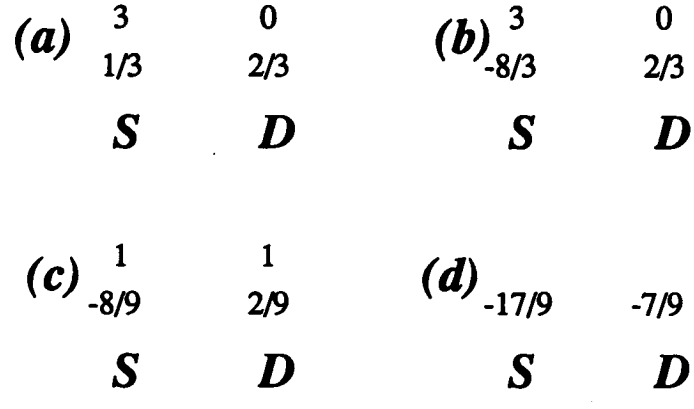

Figure 6.3: The Calculation of x_1

[1] *n.b.* $\frac{d(uv)}{dx} = v\frac{du}{dx} + u\frac{dv}{dx}$

6.6 A Recommendation for Future Development

It has been shown that generating the numerical values of the analytical derivatives of an equation set using an interpretive method is desirable in that it precludes algebraic manipulations. However, like most other methods which are available, it suffers from an inherent algorithmic inefficiency, *i.e.* the derivative of each dependent variable and each reduced equation is found with respect to each independent variable, whether it can be seen *a priori* to be zero or not. Consider, for example, the binary ideal flash problem described in appendix B, using the assignment

$$
\begin{array}{lrll}
1. & K_2 x_2 & = y_2, & K_2 \\
2. & P_2^*/P_t & = K_2, & P_t \\
3. & P_2^* x_2 & = P_2, & P_2 \\
4. & P_1 + P_2 & = P_t, & P_1 \\
5. & P_1^* x_1 & = P_1, & x_1 \\
6. & L + V & = 1, & L \\
7. & L x_1 + V y_1 & = F z_1, & V \\
8. & L x_2 + V y_2 & = F z_2, & x_2
\end{array}
$$

A computational scheme for this problem is

$$\begin{aligned} K_2 &= y_2/x_2 \\ P_t &= P_2^*/K_2 \\ P_2 &= P_2^* x_2 \\ P_1 &= P_t - P_2 \\ x_1 &= P_1/P_1^* \\ L &= F - V \end{aligned}$$

$$\begin{aligned} f_1 &= (Fz_1 - Lx_1)\, y_1 - V \\ f_2 &= (Fz_2 - Vy_2)/L - x_2 \end{aligned}$$

If Jacobi iteration is used, then each variable value is updated at the end of an iteration, and the elements of the reduced Jacobian are

$$\begin{aligned} \frac{df_1}{dV} &= \frac{\partial f_1}{\partial V} \\ \frac{df_1}{dx_2} &= 0 \\ \frac{df_2}{dV} &= \frac{\partial f_2}{\partial L}\frac{\partial L}{\partial V} + \frac{\partial f_2}{\partial V} \\ \frac{df_2}{dx_2} &= \frac{\partial f_2}{\partial x_2} \end{aligned} \tag{6.11}$$

Since the partial derivative of L with respect to V is the only derivative of a dependent variable required, most of the effort expended in calculating the derivatives using the above method would be wasted. If Gauss-Seidel iteration is used instead, then the reduced Jacobian becomes

$$\begin{aligned} \frac{df_1}{dV} &= \frac{\partial f_1}{\partial L}\frac{\partial L}{\partial V} + \frac{\partial f_1}{\partial V} \\ \frac{df_1}{dx_2} &= \frac{\partial f_1}{\partial x_1}\frac{\partial x_1}{\partial V} + \frac{\partial f_1}{\partial L}\frac{\partial L}{\partial V} + \frac{\partial f_1}{\partial x_2} \\ \frac{df_2}{dV} &= \frac{\partial f_2}{\partial L}\frac{\partial L}{\partial V} + \frac{\partial f_2}{\partial V} \\ \frac{df_2}{dx_2} &= \frac{\partial f_2}{\partial L}\frac{\partial L}{\partial x_2} + \frac{\partial f_2}{\partial y_2}\frac{\partial y_2}{\partial x_2} + \frac{\partial f_2}{\partial x_2} \end{aligned} \tag{6.12}$$

Once again slavish adherence to the algorithm described for calculating the

analytical derivatives leads to a waste of effort in that only the values of L and x_2 are dependent on that of V.

In order to determine the minimal set of derivatives required for a solution, one may use an *inheritance* graph for the equation set; that for the binary flash example is shown in figure 6.4. This digraph is a layered reordering of the signal

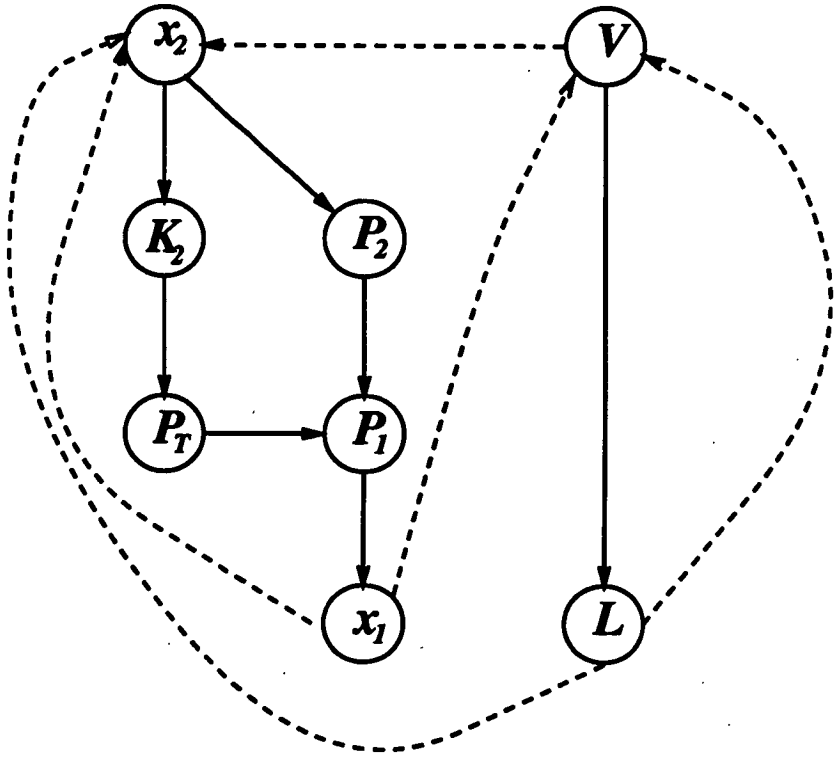

Figure 6.4: The Inheritance graph for the flash problem

flowgraph for the variables in the problem where the torn nodes are ordered first. Those nodes which represent variables which are dependent only on tear variables are ordered next; *n.b.* there must be at least one of these. The rest of the nodes are ordered in similar fashion, *i.e.* node ω is ordered in the j^{th} layer of the graph if it represents a variable whose value is dependent only on variables which have

been ordered before it, at least one of which corresponds to a vertex in the $j-1^{th}$ layer. The back edges in this digraph, which are depicted in figure 6.4 as broken arcs, show the dependency of the tear variables on the dependent variables; the cross edges depict the interdependency of the tear variables. This dependency is the key to the interpretation of the digraph.

The use of Jacobi iteration requires the derivatives of only those dependent variables which are explicit functions of independent variables, and which appear explicitly in one or more of the tear equations. More precisely, $\frac{\partial \omega}{\partial \nu}$ is required if, and only if, ν is a tear variable but ω is not, and if there is a diedge (ν, ω) and another directed edge (ω, σ) in the signal flowgraph for the variables in the problem, where ν is a tear variable, but ν and σ are not necessarily distinct. These derivatives can be identified by a depth first search of the inheritance graph for the equation set. So too the other derivatives which are required, *i.e.* the rate of change of each of the the tear equations with those torn variables which appear explicitly within it, can be identified from a search of this digraph. The necessary condition on their inclusion is that there exists a directed edge, (ν, ω), in the digraph between two torn nodes ν and ω; in this case the derivative with respect to ν of the tear equation which is matched with ω must be calculated.

The rules for identifying the necessary derivatives for use with Gauss-Seidel iteration are more complicated, and it is from these that the above graph takes its name. If there is an edge (α, β) in the inheritance graph then β is an implicit function of each of the variables which determine the value of α. If these variables are restricted to those which are independent, then, using this relationship recursively, and the fact that the nodes in the first layer of the graph are torn, it can be seen that β is a function of all of the tear variables in the

union of the of the sets of tear variables of those nodes which direct edges onto it. If β is itself a torn variable, then the derivative of the equation which is matched with it, f_β, must be found with respect to each variable which directs either a back edge or a cross edge onto β. Further, if there is a non-trivial path from some node γ, not necessarily distinct from β, which corresponds to a tear variable, x_γ, and which passes only through vertices which represent dependent variables, then the derivative of each of these variables with respect to x_γ must be calculated; *n.b.* this does not imply that each of these appears in a term in $J = \nabla_{\dot{x}}\hat{f} \times \nabla_{\dot{x}}\check{x} + \nabla_{\dot{x}}\hat{f}$. Each path of this type which passes from a node η, which represents the tear node x_η, through a node κ, which represents the dependent variable x_κ, implies that the value of $\frac{\partial \kappa}{\partial \eta}$ is required for the solution of the reduced equations.

The justification for these observations is that these paths show how the variables in the equation set vary with each other, and hence which will have nonzero derivatives with respect to each other. Returning to the problem of figure 6.4, it is seen that the values of $\{\frac{\partial K_2}{\partial x_2}, \frac{\partial P_T}{\partial x_2}, \frac{\partial P_2}{\partial x_2}, \frac{\partial P_1}{\partial x_2}, \frac{\partial x_1}{\partial x_2}, \frac{\partial L}{\partial x_2}, \frac{\partial L}{\partial V}, \frac{\partial f_1}{\partial V}, \frac{\partial f_1}{\partial x_2}, \frac{\partial f_2}{\partial V}, \frac{\partial f_2}{\partial x_2}\}$ must be calculated.

The above observations have been used to analyse a number of examples, but so far no attempt has been made to develop them algorithmically.

6.7 Summary

It was decided that the Newton Raphson method should be used to accelerate the tear equations and that the numerical values of the analytical derivatives of these equations would be calculated along with the equations themselves. As was shown in § 6.3 this can be done by adapting Ponton's [75] to torn systems. Although this was developed successfully it suffers from the defect that each dependent variable must be differentiated with respect to each independent variable, regardless of whether it is a function of it or not, and the same is true for the tear equations. The rules for a graphical analysis which circumscribes this difficulty were introduced in § 6.6, but they have not been developed as an algorithm.

In the next chapter we describe how this solution technique, and the assignment, partitioning and tearing algorithms of chapters § 4 and § 5, have been implemented as part of a mathematical modeller.

I must Create a System, or be enslav'd by another Man's;
I will not Reason and Compare: my business is to Create.
William Blake, Jerusalem

Chapter 7

The Software Implementation

7.1 Introduction

In chapter 1, it is shown that the task of producing and solving a mathematical model can be decomposed into six subtasks:

1. Find the generic statement of the problem and the data specific to the current problem.
2. Identify the necessary equations.
3. Partition the equations into a computational sequence for the problem.
4. Tear and order the equations in each partition.
5. Produce a computer program to solve the equations.

6. Solve the equations and report the results.

The theoretical basis for these features was developed in chapters 4, 5, and 6. In this chapter the practical details of implementation, such as the languages used and the maintenance and manipulation of data structures, are presented. The reason for the use of Prolog and C are given in § 7.2. A statement of the controlling algorithm for the modeller is presented in appendix E, and this is discussed along in § 7.3. The functionality of the software and the data structures which it manipulates sre discussed in this section too. In § 7.4 a sample modelling session is presented which illustrates the points discussed in § 7.3, and a summary of the chapter is presented in § 7.6.

7.2 Introduction

The modelling software was developed in a mixture of Prolog and C. All of the symbolic computation, such as equation parsing and rearrangement, was written in Prolog. This is an interpreted language and consequently it is very slow. Further, it possesses very poor numerical processing capabilities and so those tasks which are computationally intensive, *i.e.* the decomposition and the solving routines, were coded in C. In the following section we will discuss the fundamental features of Prolog which are necessary for the rest of the chapter; for a more comprehensive introduction to this language see [91].

7.2.1 Programming in Prolog

Prolog, *Prog*ramming in *Log*ic, is an interpreted, object orientated language. It is used to test the truth of simple or compound statements such as "A is a member of list B", "C is a leaf of tree D and has a value E", *etc.* It tests for this truth by comparing data structures to see if they match. If they are equal then the statement being tested is true; otherwise it is false. These data structures may be formed dynamically, or they may be contained in *rules* and *facts* which have been asserted to the Prolog database. A compound statement of rules and facts is called a *goal.*

There are three basic data types in Prolog - *atoms*, *structures* and *variables.* Atoms are constants such as integer and real numbers, or strings. For example, 1, 6.2 and 'foo' are all atoms. Structures, or *predicates*, are compound terms consisting of a *name* and a number of *arguments.* For example, in the predicate 'member(a, B)', 'member' is its name and 'a' and 'B' are its arguments. Two common types of structure are the list and the tree. Finally, variables have no value and so they can *share*, *i.e.* assume the value, of any atom or structure with which they are compared.

A Prolog program is a depth first search (DFS) of a subset of the rules and facts in the Prolog database in order to verify a series of statements; this allows one to develop massively recursive code. In general this DFS consumes a large amount of the heap assigned to the program. If a match is obtained, then very little of this memory is returned for future use; if none is found, however, the majority of this space is made available once more. Thus large Prolog programs ought to be

written so that most of their enquiries fail, rather than succeed.

7.3 A Description of the Modelling Software

The modelling software is constructed as a series of tasks which are consulted by Prolog and asserted to its database. These are retracted sequentially and an attempt is made to complete them. When no tasks remain the program stops. This consultation and manipulation is controlled by the modelling interpreter. This is defined by the predicate *program* which is shown in appendix F. Here the character '!' is referred to as the *cut*, and it commits the program to all choices made when control backtracks beyond it. As well as directing the flow of information within the system, this program cleans up the database as facts, rules and other data structures become obsolete.

Figure 7.1 is a logic flow diagram for the modelling software. Each problem type is represented by a set of equations which describe, *e.g.*, the mass and thermal balances over the system. It was shown in § 4.2.2 that this equation set may be very large and that it may contain redundant and conflicting equations. In the present system, however, each system is small and none of the equations are contradictory. Also associated with each instance of a problem type is a set of constants, *e.g.* Wilson equation parameters, fixed values and design variables; these too are described in § 4.2.2. Together these sets make up the abstract representation of a problem and they are grouped together in a file.

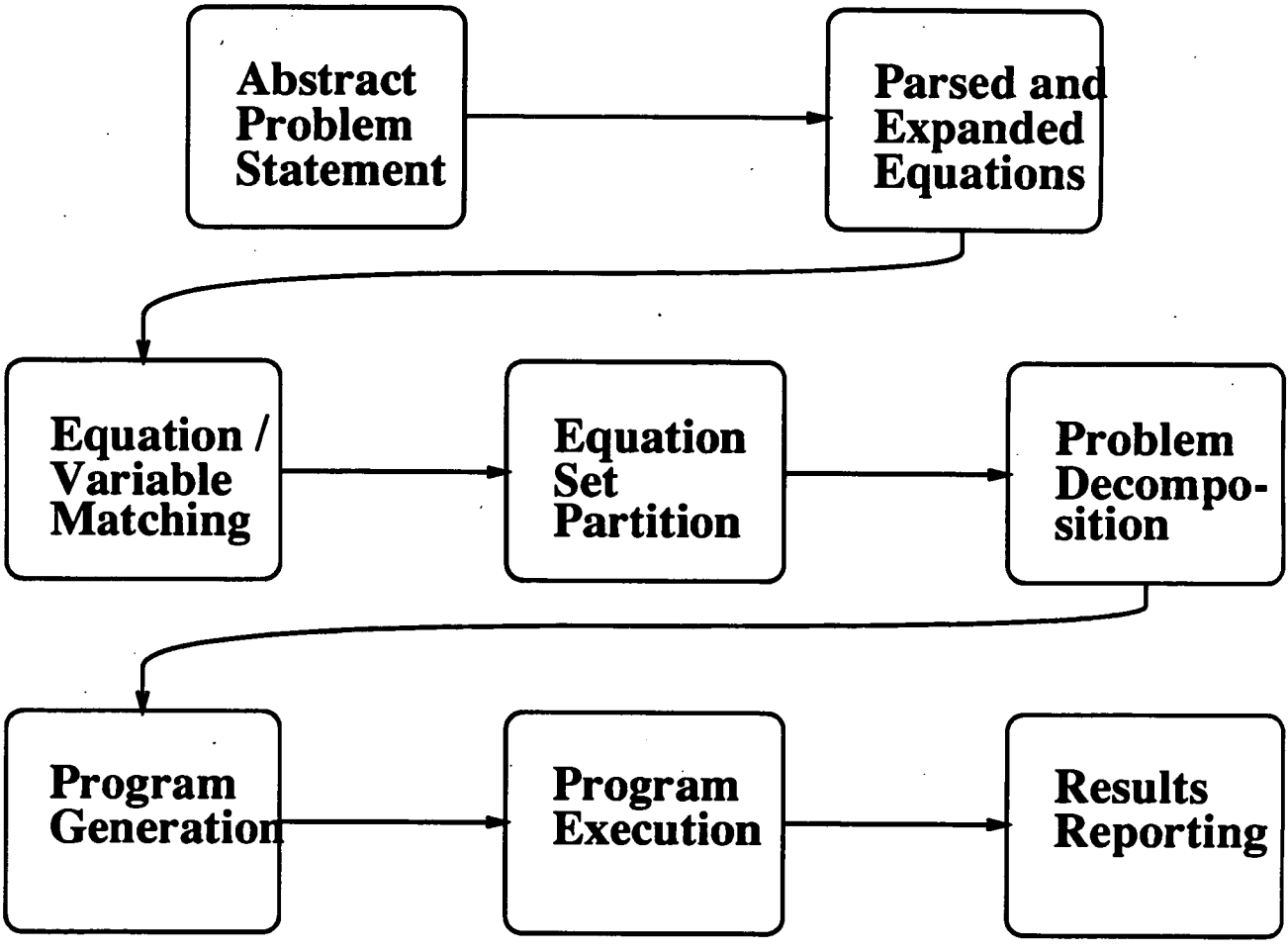

Figure 7.1: A Logic Flow Diagram for the Modelling Software

As a first step these equations and constants are consulted by Prolog and inserted into the database. In order to facilitate equation manipulation, they are then transformed from their infix form to Reverse Polish Notation; this representation was described in § 6.3. Note that there may be two types of equation present, scalar and vectorial. A scalar equation is one in which each term is a scalar; a vectorial equation is one in which at least one of the terms is a vector. Vectorial equations, *i.e.* a single term is used to denote more than one like defined scalar terms, or they may represent a number of like defined scalar equations. Regardless of their form, all vectorial equations in the database are expanded to their scalar equivalent prior to manipulation. This requires that the user be prompted for the maximum value of each index, and that vectorial unknowns and constants be replaced by their components.

At this point it is necessary to determine which unknowns appear in each equation, and in which equations each unknown appears. This is achieved by a depth first search of the parsed equation and the data are maintained as two trees, *eqn_uns* and *vars* respectively. These trees are *23_trees*, *i.e.* there are either two or three branches from each node within them. Using this form of balanced tree allows one to access any of the N leaves within the tree in between $log_3 N$ and $log_2 N$ operations. This requires that one record the lowest values accessible in the branches from one internal node and addition and deletion from the tree are non-trivial operations. For a detailed discussion of this data structure see [2].

It was seen in § 4 that a matching has to be derived between the variables in the system and the equations in which they appear. This is constructed by satisfying the goal *match*, and it is stored within Prolog as a set of facts of the

form *match(V,E)*, where V is a variable and E is an equation. Once this has been established it is used by the goal *tarjan* to identify the minimal, solvable equation subsets for the problem. These sets are maintained as a list of lists in which appear the variables associated with each subproblem and cursors for the equations to be solved for them; *n.b.* these sublists are ordered so that they correspond to a computational sequence. Lastly, a tear set for each equation subset is determined by calling the operating system for the program *bark_mot.c*. This requires that the variables in the subsets be mapped onto an array of integers and the inverse mapping is used when the results are consulted by the modelling software. The torn subsets are recorded as a list of structures of the form *tear(Vars,Tear)*. Here *Vars* is the set of dependent variables and *Tear* is the set of those which are independent.

Having analysed the equation set and thus derived a computational sequence for it, the next step is to solve the equations. This is achieved by writing two computer programs; compiling them; linking them with precompiled code and executing the object file. A logic diagram for the solving software appears in figure 7.2.

In this system the minimal equation sets are solved sequentially. If only one equation is to be solved for its solitary unknown, then this is evaluated explicitly. If, however, more than one equation is to be solved, then the solution method described in § 6.4 is employed. In either case, the equations for each subset are contained in a separate subroutine in the file *pol_eval.c*. A pointer to each routine which contains more than one equation is passed in turn to *n_raph.c* which attempts to converge the equations within it. If the attempt is successful then the next subroutine is called; otherwise an error message is written and

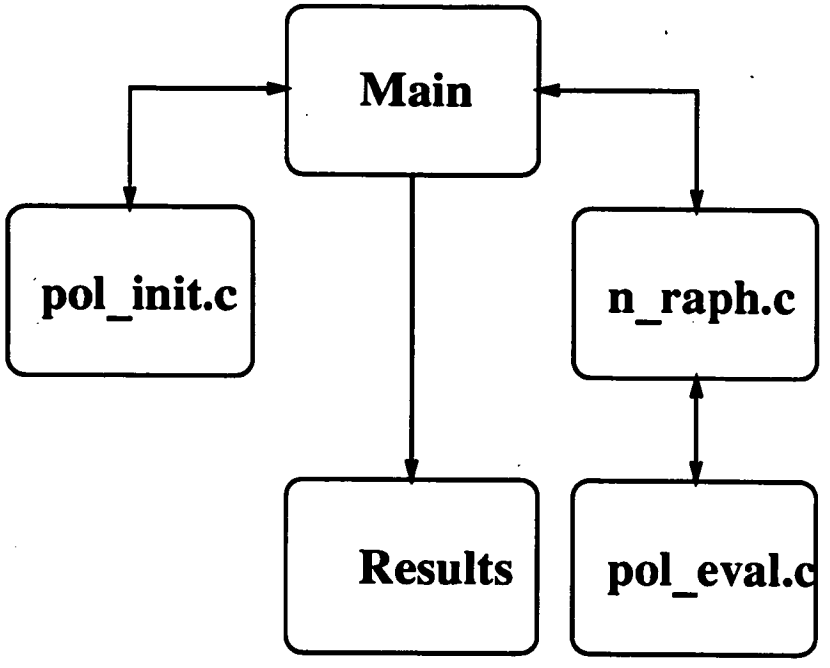

Figure 7.2: A Logic Flow Diagram for the Modelling Software

computation stops. If all the subroutines have been evaluated successfully, the solution is written to a results file. In order to minimise the amount of storage required for this computation, only as much memory as is necessary is accessed by the program. The logic for this is contained within *pol_init.c*. In particular this assigns a pointer to each subroutine in *pol_init.c* and it requisitions the space for the list of tear variables in each of them. So too this file contains the values of the constants, fixed variables and the initial guesses for the tear variables.

Finally, if the object code has been executed successfully the results are consulted by the modelling software, asserted to the Prolog database and then displayed to the user.

7.4 An Example Modelling Session

In order to illustrate the features described above, we will follow a modelling session in which a model is constructed and solved for a ternary flash problem. The components are methanol, ethanol and water, and the vapour liquid equilibrium conditions are calculated using Wilson's equations [105]. The physical property data used for the calculations are

Component	z_i	C_p	Hbase	*Delhv*
Water	0.300	75.3	-242000	40683
Ethanol	0.300	97.1	-234960	38770
Methanol	0.400	80.4	-201300	35278

where z is the mole fraction of each component in the feed stream, C_p is the specific heat capacity at constant pressure for each liquid in $kJkmol^{-1}K^{-1}$, $Hbase$ is the specific enthalpy of formation of each component at $298K$ in $kJkmol^{-1}$ and $Delhv$ is the latent heat of vapourization of each liquid, also in $kJkmol^{-1}$; *n.b.* the values of C_p and $Delhv$ vary with temperature, but they have been treated as constants here.

The Wilson equation constants, λ_{ij}, for each of these components are

Component	λ_{ij}		
Water	1.00000	0.81564	0.94934
Ethanol	0.20022	1.00000	0.60908
Methanol	0.43045	1.35386	1.00000

Lastly, the feed rate to the flash drum is assumed to be $100 Kmolhr^{-1}$ and the feed temperature is $298K$. The flash is to take place at $348.5K$ and the liquid phase mole fraction of water is to be 0.422.

The physical data were gleaned from Perry [72] and Sinnot [85] and the Wilson constants were taken from [45]. The results were checked against those obtained from PPDS.

7.4.1 The Physical and Thermodynamic Equations

Consider the flash drum represented by figure 7.3

A Prolog representation of the equations used to solve for the equilibrium conditions within it appears in figure 7.4. The structure *known_eqn*(E) represents an equation and *stat_known* is either a constant or a fixed variable. The structure *all_uns* is a list of the design variables.

```
/* The mass balance equations */

known_eqn(f=fliq+v).
known_eqn(f*z(i)=fliq*x(i)+v*y(i)).
known_eqn(sum(y(i),i)=1).
known_eqn(sum(z(i),i)=1).
```

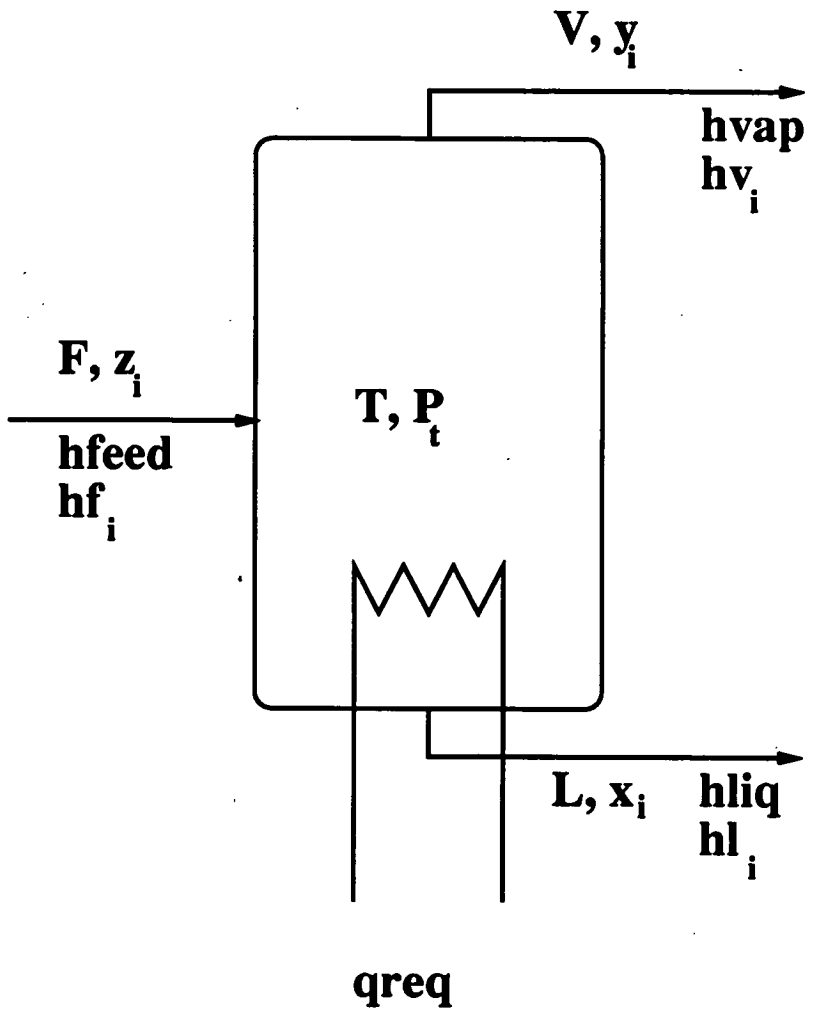

Figure 7.3: A Flash Drum

```
/* The basic equilibrium equation */

known_eqn(pt*y(i)=gamma(i)*x(i)*pstar(i)).

/* The enthalpy balance equations */

known_eqn(f*hfeed=fliq*hliq+v*hvap+qreq).
known_eqn(hfeed=sum(hf(i)*z(i),i)).
```

```
known_eqn(hliq=sum(hl(i)*x(i),i)).
known_eqn(hvap=sum(hv(i)*y(i),i)).
known_eqn(hf(i)=hbase(i)+cp(i)*(tfeed-tbase)).
known_eqn(hl(i)=hbase(i)+cp(i)*(temp-tbase)).
known_eqn(hv(i)=hl(i)+delhv(i)).

/* Lastly Antoine's equation. Gives Pstar in mm Hg using */
/* Sinnot's values if T is in Kelvin.                    */

known_eqn(log(pstar(i))=anta(i)-antb(i)/(temp+antc(i))).

/* And now the Wilson equation */

known_eqn(log(gamma(i)*w_sum(i))=w_coeff(i)).
known_eqn(w_coeff(i)=1-sum(x(j)*lambda(j, i)/w_sum(j),j)).
known_eqn(w_sum(i)=sum(x(j)*lambda(i,j),j)).

/* And now for the fundamental constants */

stat_known(anta(i)).
stat_known(antb(i)).
stat_known(antc(i)).
stat_known(lambda(i,j)).
stat_known(tfeed).
stat_known(tbase).
stat_known(cp(i)).
stat_known(hbase(i)).
```

```
stat_known(delhv(i)).

/* ... and the constants for this problem. */

stat_known(f).
stat_known(z_1).
stat_known(z_3).
stat_known(x_1).
stat_known(temp).

/*   The list of 'required' values   */

all_uns([qreq, f, fliq, v, z(i), x(i), y(i), temp, pt]).

/* Lastly, how to interpret the arrays ..... */

index_interp(i, "number of components").
index_interp(j, "number of components").
```

Figure 7.4: The Abstract Form of the Problem

7.4.2 Parsing and Expanding the Equations

Prior to manipulation, these equations must be parsed into Reverse Polish Notation. If any of them is a vectorial equation it must be expanded into its scalar form(s). The predicate *n_pol* transforms the equations to the form shown

in figure 7.5 and *vector_expand* finds their expanded

```
eqn(eqn(fliq,v,+),f,=)
eqn(eqn(f,z(i),*),eqn(eqn(fliq,x(i),*),eqn(v,y(i),*),+),=)
eqn(eqn(y(i),i,sum),1,=)
eqn(eqn(z(i),i,sum),1,=)
eqn(eqn(pt,y(i),*),eqn(eqn(gamma(i),x(i),*),pstar(i),*),=)
eqn(eqn(f,hfeed,*),eqn(eqn(eqn(fliq,hliq,*),eqn(v,hvap,*),+),
    qreq,+),=)
eqn(eqn(eqn(hf(i),z(i),*),i,sum),hfeed,=)
eqn(eqn(eqn(hl(i),x(i),*),i,sum),hliq,=)
eqn(eqn(eqn(hv(i),y(i),*),i,sum),hvap,=)
eqn(eqn(eqn(eqn(tfeed,tbase,-),cp(i),*),hbase(i),+),hf(i),=)
eqn(eqn(eqn(eqn(temp,tbase,-),cp(i),*),hbase(i),+),hl(i),=)
eqn(eqn(hl(i),delhv(i),+),hv(i),=)
eqn(eqn(pstar(i),_304311,log),eqn(eqn(eqn(temp,antc(i),+),
    antb(i),\),anta(i),<-),=)
eqn(eqn(eqn(gamma(i),w_sum(i),*),_304311,log),w_coeff(i),=)
eqn(eqn(eqn(eqn(eqn(x(j),lambda(j,i),*),w_sum(j),/),j,sum),
    1,<-),w_coeff(i),=)
eqn(eqn(eqn(x(j),lambda(i,j),*),j,sum),w_sum(i),=)
```

Figure 7.5: The Parsed Equation Set

form. Note that the predicate *eqn* has three arguments - an operator and its operands. If the operator is unary, *e.g.* log, then the right operand is a Prolog variable. The expanded equations appear in figure 7.6 where, for clarity, they appear in infix form. Each vector in the original equations is replaced by its

components in the new set. These components have the same root names as the vectors but they are suffixed by indices. The vectorial equations are replaced by the relevant number of scalar equations.

```
w_sum_3=lambda_3_3*x_3+lambda_3_2*x_2+
        lambda_3_1*x_1
w_sum_2=lambda_2_3*x_3+lambda_2_2*x_2+
        lambda_2_1*x_1
w_sum_1=lambda_1_3*x_3+lambda_1_2*x_2+
        lambda_1_1*x_1
w_coeff_3=1-(lambda_1_3*x_1/w_sum_1+lambda_2_3*x_2/w_sum_2+
        lambda_3_3*x_3/w_sum_3)
w_coeff_2=1-(lambda_1_2*x_1/w_sum_1+lambda_2_2*x_2/w_sum_2+
        lambda_3_2*x_3/w_sum_3)
w_coeff_1=1-(lambda_1_1*x_1/w_sum_1+lambda_2_1*x_2/w_sum_2+
        lambda_3_1*x_3/w_sum_3)
w_coeff_3=log(gamma_3*w_sum_3)
w_coeff_2=log(gamma_2*w_sum_2)
w_coeff_1=log(gamma_1*w_sum_1)
anta_3-antb_3/(temp+antc_3)=log(pstar_3)
anta_2-antb_2/(temp+antc_2)=log(pstar_2)
anta_1-antb_1/(temp+antc_1)=log(pstar_1)
hv_3=hl_3+delhv_3
hv_2=hl_2+delhv_2
hv_1=hl_1+delhv_1
hl_3=(temp-tbase)*cp_3+hbase_3
hl_2=(temp-tbase)*cp_2+hbase_2
```

```
hl_1=(temp-tbase)*cp_1+hbase_1
hf_3=(tfeed-tbase)*cp_3+hbase_3
hf_2=(tfeed-tbase)*cp_2+hbase_2
hf_1=(tfeed-tbase)*cp_1+hbase_1
hvap=hv_3*y_3+hv_2*y_2+hv_1*y_1
hliq=hl_3*x_3+hl_2*x_2+hl_1*x_1
hfeed=hf_3*z_3+hf_2*z_2+hf_1*z_1
fliq*hliq+v*hvap+qreq=f*hfeed
gamma_3*x_3*pstar_3=pt*y_3
gamma_2*x_2*pstar_2=pt*y_2
gamma_1*x_1*pstar_1=pt*y_1
1=z_3+z_2+z_1
1=y_3+y_2+y_1
fliq*x_3+v*y_3=f*z_3
fliq*x_2+v*y_2=f*z_2
fliq*x_1+v*y_1=f*z_1
f=fliq+v
```

Figure 7.6: The Expanded form of the Equations

7.4.3 The Variable/Equation Matching

Figure 7.7 is the matching defined by *match*. Note that in each pair the first structure is the variable which is to be solved for and a cursor to the matched equation; these structures were extracted from the Prolog database. The second structure is the infix form of the equation, and it has been included to demonstrate

that the matching is legal.

```
z_2 29 1=z_3+z_2+z_1
v 31 fliq*x_3+v*y_3=f*z_3
gamma_3 26 gamma_3*x_3*pstar_3=pt*y_3
w_coeff_1 6 w_coeff_1=1-(lambda_1_1*x_1/w_sum_1+
            lambda_2_1*x_2/w_sum_2+lambda_3_1*x_3/w_sum_3)
w_coeff_2 5 w_coeff_2=1-(lambda_1_2*x_1/w_sum_1+
            lambda_2_2*x_2/w_sum_2+lambda_3_2*x_3/w_sum_3)
w_coeff_3 4 w_coeff_3=1-(lambda_1_3*x_1/w_sum_1+
            lambda_2_3*x_2/w_sum_2+lambda_3_3*x_3/w_sum_3)
w_sum_1 3 w_sum_1=lambda_1_3*x_3+lambda_1_2*x_2+lambda_1_1*x_1
w_sum_2 2 w_sum_2=lambda_2_3*x_3+lambda_2_2*x_2+lambda_2_1*x_1
pstar_3 10 anta_3-antb_3/(temp+antc_3)=log(pstar_3)
pstar_2 11 anta_2-antb_2/(temp+antc_2)=log(pstar_2)
pstar_1 12 anta_1-antb_1/(temp+antc_1)=log(pstar_1)
hf_3 19 hf_3=(tfeed-tbase)*cp_3+hbase_3
hf_2 20 hf_2=(tfeed-tbase)*cp_2+hbase_2
x_2 32 fliq*x_2+v*y_2=f*z_2
fliq 34 f=fliq+v
y_1 33 fliq*x_1+v*y_1=f*z_1
x_3 1 w_sum_3=lambda_3_3*x_3+lambda_3_2*x_2+lambda_3_1*x_1
w_sum_3 7 w_coeff_3=log(gamma_3*w_sum_3)
hliq 23 hliq=hl_3*x_3+hl_2*x_2+hl_1*x_1
pt 28 gamma_1*x_1*pstar_1=pt*y_1
gamma_1 9 w_coeff_1=log(gamma_1*w_sum_1)
hv_3 13 hv_3=hl_3+delhv_3
```

```
hl_3 16 hl_3=(temp-tbase)*cp_3+hbase_3
hv_2 14 hv_2=hl_2+delhv_2
hl_2 17 hl_2=(temp-tbase)*cp_2+hbase_2
y_3 30 1=y_3+y_2+y_1
y_2 27 gamma_2*x_2*pstar_2=pt*y_2
gamma_2 8 w_coeff_2=log(gamma_2*w_sum_2)
hvap 22 hvap=hv_3*y_3+hv_2*y_2+hv_1*y_1
hv_1 15 hv_1=hl_1+delhv_1
hl_1 18 hl_1=(temp-tbase)*cp_1+hbase_1
qreq 25 fliq*hliq+v*hvap+qreq=f*hfeed
hfeed 24 hfeed=hf_3*z_3+hf_2*z_2+hf_1*z_1
hf_1 21 hf_1=(tfeed-tbase)*cp_1+hbase_1
```

Figure 7.7: The Expanded form of the Equations

7.4.4 The Equation Subsets

The list of minimal, solvable equation subsets is stored under the name *components* in the *parts* sector of the Prolog database. For the flash problem, it is that shown in figure 7.8.

```
| ?- recorded(dfs, components(I), _).

    I=[[hf_1,21],[hf_2,20],[hf_3,19],[hl_1,18],[hl_2,17],[hl_3,16],
      [hv_1,15],[hv_2,14],[hv_3,13],[pstar_1,12],[pstar_2,11],
```

```
[pstar_3,10],[z_2,29],[hfeed,24],[w_sum_2,2,w_sum_1,3,w_coeff_3,
4,gamma_2,8,w_coeff_2,5,gamma_1,9,w_coeff_1,6,x_3,1,w_sum_3,7,
gamma_3,26,pt,28,y_1,33,fliq,34,v,31,y_3,30,y_2,27,x_2,32],
[hvap,22],[hliq,23],[qreq,25]]
```

Figure 7.8: The Minimal Solvable equation Sets

7.4.5 The Decomposed equation Subsets

Each minimal, solvable equation subset is decomposed using the Barkley and Motard algorithm [65]. The results of this tearing are stored as a list, *partitions*, of structures of the form $tear(Vars, Tear)$. Here, $Vars$ is the set of dependent variables and $Tear$ is the set of those which are torn; *n.b.* if no variables are torn then $Tear$ is the empty list, []. In our example, only one of the minimal, solvable subsets is decomposed with a non-empty tear set. The structure stored in the Prolog database for the flash problem is displayed in figure 7.9.

```
| ?- recorded(parts, partitions(I), _).

    I=[tear([25,qreq],[]),tear([23,hliq],[]),tear([22,hvap],[]),
      tear([x_3,fliq,y_1,w_sum_2,w_sum_1,w_coeff_1,gamma_1,pt,
      w_coeff_2,gamma_2,y_2,y_3,w_coeff_3,gamma_3],[x_2,v,w_sum_3]),
      tear([24,hfeed],[]),tear([29,z_2],[]),tear([10,pstar_3],[]),
      tear([11,pstar_2],[]),tear([12,pstar_1],[]),tear([13,hv_3],[]),
      tear([14,hv_2],[]),tear([15,hv_1],[]),tear([16,hl_3],[]),
```

```
tear([17,hl_2],[]),tear([18,hl_1],[]),tear([19,hf_3],[]),
tear([20,hf_2],[]),tear([21,hf_1],[])]
```

Figure 7.9: The Decomposed Variable Subsets

7.5 Solving the Equations

7.5.1 Program Generation

A listing of the file *pol_init.c* which was written to initialise each subroutine in *pol_eval.c* appears in appendix F. The first routine in this file declares the number of subroutines in *pol_eval.c*, *i.e.* the number of equation subsets to be solved, and the number of constants and variables which appear in them. Next it declares the names of the routines in *pol_eval.c*, assigns a pointer to each of them and declares space for the array of values. Finally the values of the constants and fixed variables are set, along with the initial guesses for the tear variables. In the next subroutine a switch is declared. For each case, the number and keys of the tear variables are declared.

A listing of *pol_eval.c* appears in appendix F. The equations to be solved appear in this file. Each solvable subset is assigned to a different subroutine and there are different forms for single an multiple equations. When a single equation is to be solved it is written in infix form and calculated explicitly. Multiple equations

are written in Reverse Polish Notation and interpreted. The memory required for their evaluation is requisitioned at the start of the routine and returned to main memory at its end.

7.5.2 Reporting the Results

If the attempt to solve the equations has failed, *e.g.* they have failed to converge, then an appropriate message is printed for the user and computation stops. Otherwise, the results are read back into Prolog and they are reported on the terminal screen. The value of each design variable, constant and unknown is reported in that order. The results for our problem are contained in figure 7.10.

Variable	Value		
temp	348.5	lambda_1_1	1
x_1	0.422	lambda_1_2	0.81564
z_3	0.4	lambda_1_3	0.94934
z_1	0.3	lambda_2_1	0.20022
f	100	lambda_2_2	1
delhv_1	40683	lambda_2_3	0.60908
delhv_2	38770	lambda_3_1	0.43045
delhv_3	35278	lambda_3_2	1.35386
hbase_1	-242000	lambda_3_3	1
hbase_2	-234960	antc_1	-46.13

hbase_3	-201300	antc_2	-41.68
cp_1	75.3	antc_3	-34.29
cp_2	97.1	antb_1	3816.44
cp_3	80.4	antb_2	3803.98
tbase	298	antb_3	3626.55
tfeed	298	anta_1	18.3036
z_2	0.3	anta_2	18.9119
y_3	0.461756	anta_3	18.5875
y_2	0.311098	hvap	-179160
y_1	0.227146	hv_3	-161962
x_3	0.296584	hv_2	-191286
x_2	0.281416	hv_1	-197514
w_sum_3	0.859232	hliq	-223759
w_sum_2	0.546552	hl_3	-197240
w_sum_1	0.933093	hl_2	-230056
w_coeff_3	-0.0881325	hl_1	-238197
w_coeff_2	-0.351091	hfeed	-223608
w_coeff_1	0.296069	hf_3	-201300
v	62.611	hf_2	-234960
qreq	-2.77727e+06	hf_1	-242000
pt	785.701	gamma_3	1.06565
pstar_3	1147.91	gamma_2	1.28793
pstar_2	674.395	gamma_1	1.44097
pstar_1	293.49	fliq	37.389

Figure 7.10: The Solution of the Equations

7.6 Summary

In this chapter we discussed the way in which the modelling software transforms an abstract statement of a problem, *i.e.* a generic set of equations and variables, into a computer program which it executes in order to solve it. This is done by parsing and expanding the equations, finding a matching between them and the unknowns and then partitioning the equation set into its minimal, solvable subsets. Next it finds a tear set of the variables for each subset of size greater than one. Lastly, it constructs and executes a computer program to solve the equations, and reports the results to the user.

Life is the art of drawing sufficient conclusions from insufficient premises

Samuel Butler

Chapter 8

Conclusions and Recommendations for Future Work

8.1 Recommendations for Future Work

Prolog is an interpreted language and thus its execution is very slow; the rate of model production would be improved by rewriting all of the algorithmic tasks such as parsing and matching in a procedural language such as C. Four further technical improvements are desirable. Firstly its scope for problem formulation would be enhanced greatly by the ability to use and solve differential equations. These may be maintained in a database in the same way that algebraic equations are at present, although some new system definitions would be necessary. Secondly, the software should be made more user friendly. One way in which this might be achieved is by supplying a menu and icon driven graphical interface which the user could manipulate instead of writing a file.

Thirdly, an intelligent front end should be written which can, in conjunction with the graphical interface proposed above, construct the abstract problem statement. This should be a frame based system in which the top level slots are used to define the types of equation to be solved; these equations ought to be maintained in a data base. As an example, if a user were to wish to solve a reactor problem then clicking on a reactor icon should prompt the system to enquire as to the type of reactor, its heat transfer characteristics, the reaction order, *etc.* As these slots are filled the corresponding equations ought to be retrieved from the database and collated in an 'active' file, thus constructing the abstract problem statement. Such an approach would provide the opportunity to make approximations, *e.g.* by assuming the specific heat capacities were constant over a range of temperatures, and to relate these to the more exact model.

Lastly, in its current state, the modelling system makes no use of the knowledge contained in other software. This should be changed so that, it can consult other databases and external programs, *e.g.*, PPDS for physical properties, or an expert system for a choice of equation of state.

8.2 Conclusions

The requirements of a mathematical modelling system were investigated in § 1. Some definitions of model optimality were considered but it was shown that, although a qualitative comparison may be made between formulations, it is not possible to provide a meaningful, precise definition of optimality.

In § 2 it was shown that, in the general case, we can place some necessary conditions on an equation set for it to have a unique, non-trivial solution. Next the desirability of establishing an output set was established, and the equivalence of graph, matrix and equation partitioning was demonstrated. Choosing an output set is a check on structural singularity and a step towards partitioning an equation set; the strong components of a directed graph correspond to the minimal diagonal blocks of a block lower triangular matrix and the minimal, solvable subsets of an equation set. Lastly optimality was considered once more, with respect to the selection of tear sets. A good definition for this proved elusive and so too it was shown that, for many numerical methods, the reduction in effort required per iteration for a torn system is insignificant.

Techniques for output selection, matrix partitioning and decomposition were examined in § 3. Dinic's maximal flow algorithm [20] is the best available method for output selection and Tarjan's depth first search [94] is the optimal formulation for matrix partitioning. No comprehensive characterisation of fill-in in unsymmetric matrices has been developed, but that for symmetric matrices is well understood; establishing the minimum fill-in for either type of matrix is an NP-complete problem. Barkley and Motard's algorithm [9] is the best available for decomposition.

It was demonstrated in § 4 that a square, solvable subset of equations which describes a particular instance of a generic problem can be identified by finding a maximum matching between the variables and equations and then partitioning the equation set. This can be achieved most efficiently by using the techniques selected in § 3. Further, those equations which are not matched with a variable may be candidates for replacing any redundant equations which are identified in

the formulation of the specific problem.

A minimum cardinality separator for a reduced signal flowgraph is also one for the bipartite digraph from which it was derived; this result appears in § 5. The signal flowgraph can be formed in time which is linear in the number of arcs in the digraph; the tear set can be found in time which is quartic in the number of its nodes. The numerical values of the analytical derivatives of the reduced equations in a torn system can be calculated simultaneously with those of the reduced equations. Rules for these calculations are presented in § 6 and a prescription for an improvement to the method is given.

Software has been developed which transforms an abstract problem statement into a mathematical model, and then realises this as a simulation. This has been demonstrated on a sample problem but some improvements are possible. In particular, greater power would be achieved by supplying an intelligent front end which can formulate the problem statement interactively with the user; by rewriting the algorithmic tasks in a procedural language such as C; adding a user friendly interface; and broadening the range of application of the software by enabling it to solve differential equations.

Bibliography

[1] A.V. Aho, M.R. Garey, and J.D. Ullman. The transitive reduction of a directed graph. *SIAM. J. Comput.*, 1(2):131–137, 1972.

[2] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *Data Structures and Algorithms.* Computer Science and Information Processing. Addison-Wesley, 1983.

[3] A. C. Aitken. On Bernoulli's numerical solution of algebraic equations. *Proceedings of the Royal Society of Edinburgh*, 46:289–305, 1925.

[4] A. C. Aitken. Studies in Practical Mathematics. V. On the iterative solution of a system of linear equations. *Proceedings of the Royal Society of Edinburgh*, 63:52–60, 1950.

[5] A L. Apostel. *Towards the formal study of models in the non-formal sciences*, page 1. Reidel Pub. Co., 1961.

[6] R. Aris. *Mathematical Modelling Techniques.* Pitman, 1978.

[7] A X.J.R. Avula, editor. *Proceedings of the first International conference on mathematical modelling.* University of Missouri, 1977.

[8] R. Banares-Alcantara and A.W. Westerberg. Development of an expert system for physical property predictions. *Comput. Chem. Eng.*, 9(2):127–149, 1985.

[9] R. W. Barkley and R. L. Motard. Decomposition of nets. *The Chemical Engineering Journal*, 3:265–275, 1973.

[10] J. G. P. Barnes. An algorithm for solving non-linear equations based on the secant method. *Computer Journal*, 8:66–72, 1965.

[11] C. Berge. Some classes of perfect graphs. In F. Harary, editor, *Graph Theory and Theoretical Physics*, pages 155–165, London, 1967. Academic Press.

[12] K. E. Bett, J. S. Rowlinson, and G. Saville. *Thermodynamics for Chemical Engineers*, page 137. The Athlone Press, 1975.

[13] C. G. Broyden. A class of methods for solving nonlinear simultaneous equations. *Math. Comp.*, 19:577–593, 1965.

[14] C. G. Broyden. Quasi-newton methods and their application to function minimisation. *Math. Comp.*, 21:368–381, 1967.

[15] R. H. Cavett. Application of numerical methods to the convergence of simulated processes involving recycle loops. Preprint 04-63, American Petroleum Institute, May 1963.

[16] J. H. Christensen and D. F. Rudd. Structuring design computations. *A.I.Ch.E. Journal*, 15(1):94–100, 1969.

[17] J. F. Cordoba. A linear algorithm for nonredundant decompositions. *Computers and Chemical Engineering*, 12(1):105–107, 1988.

[18] C. M. Crowe and M. Nishio. Covergence promotion in the simulation of chemical processes - the general dominant eigenvalue method. *A.I.Ch.E. Journal*, 21(3):528–533, 1975.

[19] E. W. Dijkstra. A note on two problems in connection with graphs. *Numerische Math.*, 1:269–271, 1959.

[20] E. A. Dinic. Algorithm for solution of a problem of maximum flow in a network with power estimation. *Soviet Math. Dokl.*, 11(5):1277–1280, 1970.

[21] A. T. Doig. Reactors and reaction mechanisms. In *The Design of a Chemical Heat Pipe, Undergraduate Design Report.* Dept. Chem. Eng., University of Edinburgh, 1985.

[22] I. S. Duff. On permutations to block triangular form. *J. Inst. Math. Applic.*, 19:339–342, 1977.

[23] I. S. Duff. On algorithms for obtaining a maximum traversal. *ACM Transactions on Mathematical Software*, 7(3):315–330, 1981.

[24] I. S. Duff and J. K. Reid. An implemantation of Tarjan's algorithm for the block triangularization of a matrix. *ACM Transactions on Mathematical Software*, 4(2):137–147, 1978.

[25] I. S. Duff and J. K. Reid. Some design features of a sparse matrix code. *ACM Transactions on Mathematical Software*, 5(1):18–35, 1979.

[26] I.S. Duff, J.K. Reid, and A.M. Erisman. *Direct Methods for Sparse Matrices.* John Wiley, 1988.

[27] J. Edmonds. Paths, trees and flowers. *Canadian Journal of Mathematics*, 17:449–67, 1965.

[28] J. Edmonds and E. Johnson. Matching: A well solved class of integer linear programs. In Hmm, editor, *Combinatorial Structures and their applications*, pages 89–92, 1970.

[29] A.M. Erisman, R.G. Grimes, J.G. Lewis, and W.G Poole, Jr. A structurally stable modification of Hellerman-Rarick's P^4 algorithm for reordering unsymmetric sparse matrices. *SIAM J. Numer. Anal.*, 22:369–385, 1985.

[30] S. Even. *Graph Algorithms.* Pitman, 1979.

[31] L.R. Ford, Jr. and D.R. Fulkerson. Maximal flow through a network. *Canadian Journal of Mathematics*, 8:399–404, 1956.

[32] G. J. Forder and H. P. Hutchison. The analysis of chemical plant flowsheets. *Chemical Engineering Science*, 24:771–785, 1969.

[33] H. Gabow. An efficient implementation of Edmonds' maximum matching algorithm. Technical Report Technical Report 31, Stanford University Computer Science Dept., June 1972.

[34] D. Gale and L. S. Shapley. College admissions and the stability of marriage. *American Mathematical Monthly*, 69:9–14, 1962.

[35] A. George and J. W. H. Liu. The evolution of the minimum degree ordering algorithm. *SIAM review*, 31(1):1–19, 1989.

[36] A. Gibbons. *Algorithmic Graph Theory*, chapter 5. Cambridge University Press, 1987.

[37] P. Hall. On representatives of subsets. *J. London Math. Soc.*, 10:26–30, 1935.

[38] F. Harary. The determinant of the adjacency matrix of a graph. *SIAM Review*, 4(3):202–210, 1962.

[39] L. Haskins and D. J. Rose. Toward characterisation of perfect elimination digraphs. *SIAM Journal on Computing*, 2(4):217–224, 1973.

[40] E. Hellerman and D. Rarick. Reinversion with the preassigned pivot procedure. *Mathematical Programming*, 1:195–216, 1971.

[41] E. Hellerman and D. Rarick. The partitioned preassigned pivot procedure. In D. J. Rose and R. A. Willoughby, editors, *Sparse Matrices and their Applications*, pages 67–76, 1972.

[42] E.J. Henley and E.M. Rosen. *Material and Energy Balance Computations*, page 110. Chemical Engineering Outlines. John Wiley, 1969.

[43] D. M. Himmleblau. Decomposition of large scale systems-1. Systems composed of lumped parameter elements. *Chemical Engineering Science*, 21:425–438, 1966.

[44] D. M. Himmleblau. Decomposition of large scale systems-2. Systems containing nonlinear elements. *Chemical Engineering Science*, 21:883–895, 1966.

[45] M. Hirata, S. Ohe, and K. Nagamaha. *The Computer Aided Data Book of Vapour-Liquid Equilibria.* Kodansha, Elsevier Scientific Publishing Company, 1975.

[46] J. E. Hopcroft and R. M. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM Journal on Computing*, 2(4):225–231, 1973.

[47] D. Hutton. *Knowledge Based Flowsheet Modelling for Chemical Process Design.* PhD thesis, University of Edinburgh, 1990.

[48] R. W. Irving and P. Leather. The complexity of counting stable marraiges. *SIAM Journal on Computing*, 15(3):655–, 1986.

[49] W. R. Johns. Mathematical considerations in preparing general-purpose computer programs for the design or simulation of chemical processes. In *The Use of Computers in the design of Chemical plants*, Florence, 1970.

[50] D. J. Kleitman. A note on perfect elimination digraphs. *SIAM. J. Comput.*, 3(4):280–282, 1974.

[51] E. Kreysig. *Advanced Engineering Mathematics*, page 807. Wiley, 1979.

[52] A W.H. Leatherdale. *The role of analogy, model and metaphor in science.* Elsevier, 1974.

[53] W. Lee, J. H. Christensen, and D. F. Rudd. Design variable selection to simplify process calculations. *A.I.Ch.E. Journal*, 12(6):1104–1110, 1966.

[54] W. Lee and D. F. Rudd. On the ordering of recycle calculations. *A.I.Ch.E. Journal*, 12(6):1184–1190, 1966.

[55] M. J. Leigh. *A computer flowsheeting programme incorporating algebraic analysis of the problem structure.* PhD thesis, University of London, 1973.

[56] K. Levenberg. A method for the solution of certain non-linear problems in least squares. *Quart. Appl. Math.*, 2:164–168, 1944.

[57] T. D. Lin and R. S. H. Mah. Hierarchical partition-a new optimal pivoting algorithm. *Mathematical Programming*, 12:260–278, 1977.

[58] J. W. H. Liu. A compact row storage scheme for cholesky factors using elimination trees. *ACM Transactions on Mathematical Software*, 12(2):127–148, 1986.

[59] J. W. H. Liu. A graph partitioning problem by node separators. *ACM Transactions on Mathematical Software*, 15(3):198–219, 1989.

[60] J.W.H. Liu. Equivalent sparse matrix reordering by elimination tree rotations. *SIAM. J. Stat. Comput.*, 9(3):424–444, 1988.

[61] H. M. Markowitz. The elimination form of the inverse and its application to linear programming. *Management Science*, 3(3):255–269, 1957.

[62] D. W. Marquardt. A method for least squares estimation of non-linear parameters. *SIAM Journal*, 11:431–441, 1963.

[63] J. M. Montagna and O. A. Iribarren. Optimal computation sequence in the simulation of chemical plants. *Computers and Chemical Engineering*, 12(1):71–79, 1988.

[64] R. L. Motard, M. Shacham, and E. M. Rosen. Steady state chemical process simulation. *A.I.Ch.E. Journal*, 21(3):417–436, 1975.

[65] R. L. Motard and A. W. Westerberg. Exclusive tear sets for flowsheets. *A.I.Ch.E. Journal*, 27(5):725–732, 1981.

[66] C. L. N. Murthy and A. Hussain. An efficient tearing algorithm based on minimum sum of weights. *Computers and Chemical Engineering*, 7(2):133–136, 1983.

[67] O. Orbach and C. M. Crowe. Convergence promotion in the simulation of chemical processes with recycle - the Dominant Eigenvalue Method. *The Canadian Journal of Chemical Engineering*, 49:509–513, 1971.

[68] C.C. Pantelides. The consistent initialisation of differential-algebraic systems. *SIAM J. Sci. Stat. Comput.*, 9(2):213–231, 1988.

[69] W. R. Paterson. A new method for solving a class of non-linear equations. *Chemical Engineering Science*, 41:1935, 1986.

[70] W. R. Paterson. On preferring iteration in a transformed variable to the method of successive substitution. *Chemical Engineering Science*, 41:601, 1986.

[71] Ding-Yu Peng and D. B. Robinson. A new two constant equation of state. *Ind. Eng. Chem (Fundamentals)*, 15(1):59–64, 1976.

[72] R. H. Perry and C. H. Chilton. *The Chemical Engineers' Handbook.* McGraw-Hill, 5^{th} edition, 1974.

[73] T. K. Pho and L. Lapidus. Topics in Computer-Aided Design: Part 1: An optimal tearing algorithm for recycle streams. *A.I.Ch.E. Journal*, 19(6):1170–1181, 1973.

[74] George Polya. *How to solve it.* Penguin, 2^{nd} edition, 1990.

[75] J. W. Ponton. The numerical evaluation of analytical derivatives. *Computers and Chemical Engineering*, 6:331–333, 1982.

[76] R. H. Rand. *Computer Algebra in Applied Mathematics: An Introduction to Macsyma.* Research Notes in Mathematics (94). Pitman, 1984.

[77] Jerry Rayna. *REDUCE - A System for Computer Algebra.* Springer, 1987.

[78] D. J. Rose. A graph-theoretic study of the numerical solution of sparse positive definite systems of linear equations. In R. Read, editor, *Graph Theory and Computing*, pages 183–217, New York, 1971. Academic Press.

[79] D. J. Rose and J. R. Bunch. The role of partitioning in the numerical solution of sparse systems. In D. J. Rose and J. R. Bunch, editors, *Sparse Matrix Computations*, 1972.

[80] D. J. Rose and R. E. Tarjan. Algorithmic aspects of vertex elimination on directed graphs. *SIAM J. Appl. Math.*, 34(1):176–197, 1978.

[81] S. Sahni. Computationally related problems. *SIAM. J. Comput.*, 3(4):262–279, 1974.

[82] R. W. H. Sargent. The decomposition of systems of procedures and algebraic equations. In G. A. Watson, editor, *Numerical Analysis-Proceedings, Biennial Conference, Dundee*, pages 158–178. Springer-Verlag, 1977. Lecture Notes in Mathematics, 630.

[83] R. W. H. Sargent and A.W. Westerberg. SPEED-UP in chemical engineering design. *Trans. Inst. Chem. Engrs.*, 42:190–197, 1964.

[84] R. Schreiber. A new implementation of sparse gaussian elimination. *ACM Transactions on Mathematical Software*, 8(3):256–276, 1982.

[85] R. K. Sinnot. *Chemical Engineering, Volume 6.* Pergamon, 1983.

[86] J. M. Smith and H. C. Van Ness. *Introduction to Chemical Engineering Thermodynamics*, page 295. M^{c}Graw Hill, 3rd edition, 1981.

[87] J.M. Smith. *Models in Ecology.* Cambridge University Press, 1974.

[88] S. Soylemez and Seider W, D. A new technique for precedence ordering chemical process equation sets. *A.I.Ch.E. Journal*, 19(5):934–942, 1973.

[89] M. A. Stadtherr, W. A. Gifford, and L. E. Scriven. Efficient solution of sparse sets of design equations. *Chemical Engineering Science*, 29:1025–1034, 1974.

[90] M. A. Stadtherr and E. S. Wood. Sparse matrix methods for equation-based chemical process flowsheeting-1: Reordering phase. *Computers and Chemical Engineering*, 8(1):9–18, 1984.

[91] L. Sterling and E. Shapiro. *The Art of Prolog: Advanced Programming Techniques.* MIT Press, 1986.

[92] D. V. Steward. On an approach to techniques for the analysis of the structure of large systems of equations. *SIAM review*, 4(4):321–342, 1962.

[93] D. V. Steward. Partitioning and tearing systems of equations. *J. SIAM Numer. Anal. Ser. B*, 2(2):345–365, 1965.

[94] R. E. Tarjan. Depth-first search and linear algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[95] R. E. Tarjan. Enumeration of the elementary circuits of a directed graph. *SIAM Journal on Computing*, 2(3):211–216, 1973.

[96] J. C. Tiernan. An efficient search algorithm to find the elementary circuits of a graph. *Communications of the ACM*, 13(12):722–726, 1971.

[97] W.F. Tinney and J.W. Walker. Directy solutions of sparse network equations by optimally ordered triangular factorization. *Proc. IEEE*, 55(11):1801–1809, 1967.

[98] R. S. Upadhye and E. A. Grens. Selection of decompositions for chemical process simulation. *A.I.Ch.E. Journal*, 21(1):136–143, 1975.

[99] J. H. Wegstein. Accelerating convergence of iterative processes. *Comm. ACM*, 1(6):9–13, 1958.

[100] H. Weinblatt. A new search algorithm for finding the simple cycles of a finite directed graph. *ACM Journal*, 19(1):43–56, 1972.

[101] A. W. Westerberg. Private communication.

[102] A. W. Westerberg and F. E. Edie. Computer-Aided Design, Part 1: Enhancing convergence properties by the choice of output variable assignments in the solution of sparse equation sets. *Chemical Engineering*, 2:9–15, 1971.

[103] A. W. Westerberg and F. E. Edie. Computer-Aided Design, Part 2: An approach to convergence and tearing in the solution of sparse equation sets. *Chemical Engineering*, 2:17–24, 1971.

[104] J. H. Wilkinson. *The Algebraic Eigenvalue Method*, page 36. Clarendon Press, 1965.

[105] G. M. Wilson. *J. Am. Chem. Soc.*, 86:127, 1964.

[106] M. Yannakakis. Computing the minimum fill-in is NP complete. *SIAM J. Alg. Disc. Methods*, 2(1):77–79, 1981.

Appendix A

The Operations Count for LU Decomposition

Consider some $N \times N$ matrix A and its factors, L and U, defined by

$$A = L\,U \tag{A.1}$$

and such that L is a lower triangular matrix with ones on its diagonal and U is upper triangular. Let there be γ_1 nonzeros below the first diagonal element of A and ρ_1 non-zero elements to the right of it in the first row. In order to calculate the elements in the first column of L, γ_1 divisions are necessary and at most another $\gamma_1 * \rho_1$ entries in A must be altered. Each of these alterations requires one subtraction and one multiplication. Let A^1 be the matrix formed by these operations, and extend this notation so that A^k is the matrix formed by the first $k-1$ sets of operations, γ_k is the number of non-zero entries below the k^{th} diagonal entry in A^k, and ρ_k is the number of non-zeros to the right of this element. The total number of operations required to form the elements of L and U is thus

$$d_{sum} = \sum_{k=1}^{k=N} \gamma_k \qquad \text{Divisions}$$

$$m_{sum} = \sum_{k=1}^{k=N} \gamma_k\ \rho_k \qquad \text{Multiplications}$$

$$a_{sum} = \sum_{k=1}^{k=N} \gamma_k\ \rho_k \qquad \text{Additions}$$

In the worst case, i.e. A is full or becomes so, $\gamma_k = \rho_k = N - k$ and so $O(N^3)$ operations are required in all.

Appendix B

A Binary Ideal Flash Problem

Consider the flash drum shown in figure B.1. The feedrate of the binary mixture into the drum is F, the vapour rate produced is V and the liquid flowrate out of the drum is L. The mole fraction of the i^{th} component is z_i in the feed, y_i in the vapour product and x_i in the liquid product. The vapour pressure of pure component i at the system temperature T is P_i^*, and its partial pressure is P_i. K_i is the Henry's Law constant for the i^{th} component at T and P_t, the system presure.

The overall mass balance on the drum is

$$F = L + V \tag{B.1}$$

the balance on each component is

$$Fz_i = Lx_i + Vy_i \tag{B.2}$$

and the definitions of x_i, y_i and z_i give

$$l\sum_{i=2}^{i=1} x_i = 1 \tag{B.3}$$

$$\sum_{i=2}^{i=1} y_i = 1 \tag{B.4}$$

$$\sum_{i=2}^{i=1} z_i = 1 \tag{B.5}$$

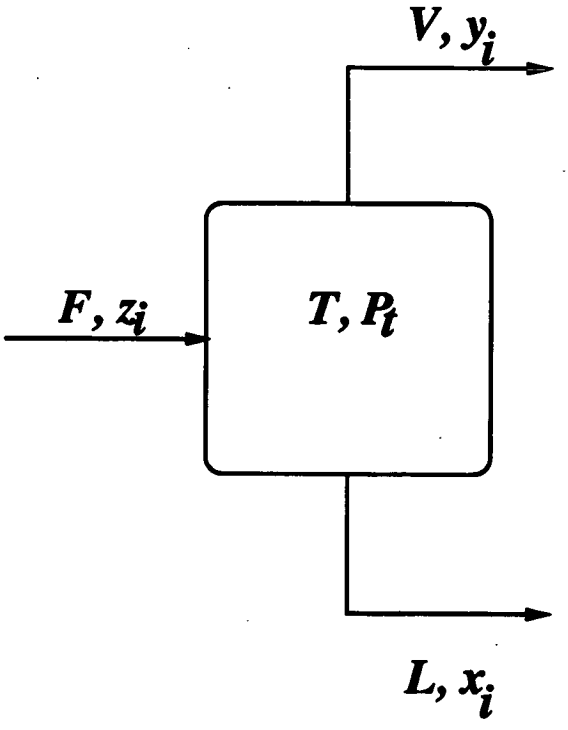

Figure B.1: A Flash Drum

The total pressure of the system is the sum of the partial pressures of the components

$$P_1 + P_2 = P_t \tag{B.6}$$

and the partial pressure of each component is related to its pure vapour pressure at the system temperature by

$$P_i = P_i^* x_i \tag{B.7}$$

The Henry's Law constant for each component relates its mole fraction in a vapour phase to that in a liquid phase with which it is in equilibrium

$$y_i = K_i x_i \tag{B.8}$$

This constant is a function of temperature and pressure and it is defined by

$$K_i = \frac{P_i^*}{P_t} \tag{B.9}$$

If T and y_1 are known, then specifying F and z_1 allows one to solve for all of the other variables. There are ten unknowns in the equation set and so ten of the equations above must be used to provide their values; *n.b.* only five of

equations B.1- B.5 may be used. One legal assignment for this problem is

$$
\begin{array}{llcll}
1. & z_1 + z_2 & = & 1, & z_2 \\
2. & Lx_1 + Vy_1 & = & Fz_1, & L \\
3. & Lx_2 + Vy_2 & = & Fz_2, & x_2 \\
4. & L + V & = & 1, & V \\
5. & y_1 + y_2 & = & 1, & y_2 \\
6. & K_2 x_2 & = & y_2, & K_2 \\
7. & P_2^* / P_t & = & K_2, & P_t \\
8. & P_1 + P_2 & = & P_t, & P_1 \\
9. & P_1^* x_1 & = & P_1, & x_1 \\
10. & P_2^* x_2 & = & P_2, & P_2
\end{array}
$$

where the variable after each equation is the variable for which it is to be solved.

Appendix C

The Dissociation of Water

The equilibrium constant, K, for a reaction in an ideal gas mixture is a function of temperature, T, alone. If there are M such competing reactions, then the M equilibrium constants K_j are given by

$$K_j = \mathcal{F}_j(T), \quad j = 1, 2, \cdots, M \tag{C.1}$$

Alternatively, the equilibrium constants can be calculated from the partial pressures of the gases in the mixture. If there are N components, each of which has a partial pressure P_i, taking part in these reactions, and if ν_{ij} is the stoichiometric coefficient of the i^{th} component in the j^{th} reaction, then, K_j may be found from

$$K_j = \prod_{i=1}^{1=M} P_i^{\nu_{ij}}, \quad j = 1, 2, \cdots, M \tag{C.2}$$

The partial pressure of each component is the product of its mole fraction in the mixture, y_i, and the total pressure, P_t,

$$P_i = P_t y_i, \quad i = 1, 2, \cdots, N \tag{C.3}$$

In turn, the i^{th} mole fraction is the ratio of the number of moles of component i, n_i, to the total number of moles, n_t,

$$y_i = \frac{n_i}{n_t} \quad i = 1, 2, \cdots, N \tag{C.4}$$

and n_t is defined by

$$n_t = \sum_{i=1}^{i=N} n_i \tag{C.5}$$

If n_{i0} is the number of moles of compound i present initially, and if ϵ_j is the extent of the j^{th} reaction [42], then

$$n_i = n_{i0} + \sum_{i=1}^{i=N} \nu_{ij}\epsilon_j \quad i = 1, 2, \cdots, N \tag{C.6}$$

The conservation of mass in a reacting system requires that, in the absence of radioactive decay, the initial and final amounts of each element present be equal. Thus, if α_{ik} is the number of moles of element k in component i, and if there are R elements present, then

$$\sum_{i=1}^{1=N} \alpha_{ik}(n_{i0} - n_i) = 0, \quad k = 1, 2, \cdots, R \tag{C.7}$$

The total pressure and the temperature of the system, its volume, V, and the total number of moles of gas present are related by the ideal gas equation of state

$$P_t\, V = n_t\, R\, T \tag{C.8}$$

where R is the universal gas constant. If the system is closed and the initial mass of each component is known, then Duhem's Theorem [86] states that its equilibrium state is specified by fixing the values of two independent variables; *n.b.* these variables may be intensive or extensive.

When water dissociates at high temperature, four independent reactions take place,

$$H_2O \rightleftharpoons H_2 + \frac{1}{2}O_2 \tag{C.9}$$

$$H_2O \rightleftharpoons \frac{1}{2}H_2 + OH \tag{C.10}$$

$$H_2 \rightleftharpoons 2H \tag{C.11}$$

$$O_2 \rightleftharpoons 2O \tag{C.12}$$

If the gases are at high temperature but low pressure, the compression factor for the mixture is almost unity [12], and so the mixture can be assumed to be ideal. Backsubstitution of equations C.4 - C.2 shows that altering the system pressure at any temperature alters the equilibrium distribution of products, *i.e.*

$$K_j = \{(\frac{P_t}{n_t})^{\sum_{i=1}^{i=N} \nu_{ij}}\} \prod_{i=1}^{i=N} n_i^{\nu_{ij}}, \quad j = 1, 2, \cdots, M \tag{C.13}$$

If the initial amount of each of the six components is known, fixing the temperature and pressure of the system specifies its equilibrium

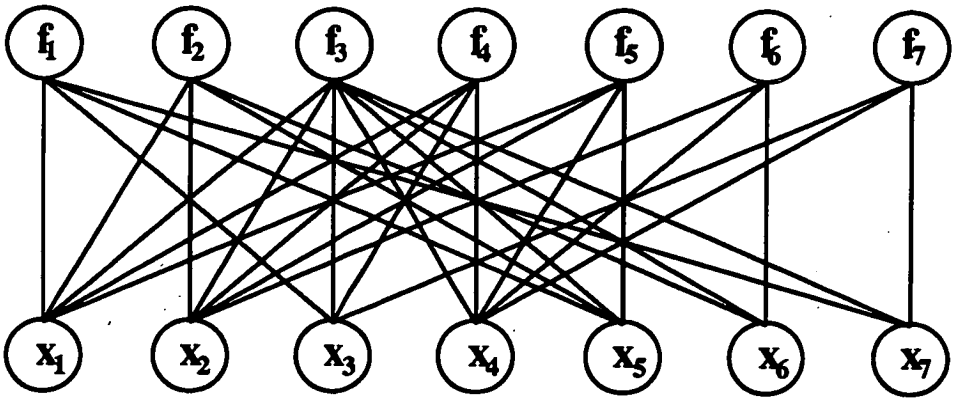

Figure C.1: The Graph of the Dissociation Equations

state. Then, following the solution of equation C.1 for each reaction, equations C.5, C.7and C.13 can be solved simultaneously and the results substituted forward to give the equilibrium conditions. The graph of this equation set is shown in figure C.1. Here equations f_1 and f_2 are the molar balances on monotonic oxygen and hydrogen respectively, f_3 is the overall molar balance, and the remaining four are the equilibrium equations for the reactions. Variables x_1, x_2 and x_3 are the equilibrium molar amounts of water, molecular hydrogen and molecular oxygen and x_5, x_6 and x_7 are those hydroxyl ions, and atomic hydrogen and oxygen respectively; x_4 is the total number of moles present at equilibrium. The incidence matrix for one ordering of the equations is shown in figure C.2.

	X1	X3	X7	X6	X2	X5	X4
f5	x				x	x	x
f4	x	x			x		x
f7		x	x				x
f6				x	x		x
f2	x			x	x	x	
f1	x	x	x			x	
f3	x	x	x	x	x	x	x

Figure C.2: The Incidence Matrix for the Dissociation Equations

If instead the equations are to be solved without algebraic substitution, then the incidence matrix for one ordering of these equations and variables is shown in figure C.3. Here the generic equations have been expanded in the order in which they appear above, and the variables are mapped onto the array x by

$$
\begin{array}{llllllllll}
P_{H_2O} & \rightarrow & x_1 & y_{H_2O} & \rightarrow & x_7 & n_{H_2O} & \rightarrow & x_{13} & P_t \rightarrow x_{19} \\
P_{H_2} & \rightarrow & x_2 & y_{H_2} & \rightarrow & x_8 & n_{H_2} & \rightarrow & x_{14} & n_t \rightarrow x_{20} \\
P_{O_2} & \rightarrow & x_3 & y_{O_2} & \rightarrow & x_9 & n_{O_2} & \rightarrow & x_{15} & \\
P_{OH} & \rightarrow & x_4 & y_{OH} & \rightarrow & x_{10} & n_{OH} & \rightarrow & x_{16} & \\
P_H & \rightarrow & x_5 & y_H & \rightarrow & x_{11} & n_H & \rightarrow & x_{17} & \\
P_O & \rightarrow & x_6 & y_O & \rightarrow & x_{12} & n_O & \rightarrow & x_{18} &
\end{array}
$$

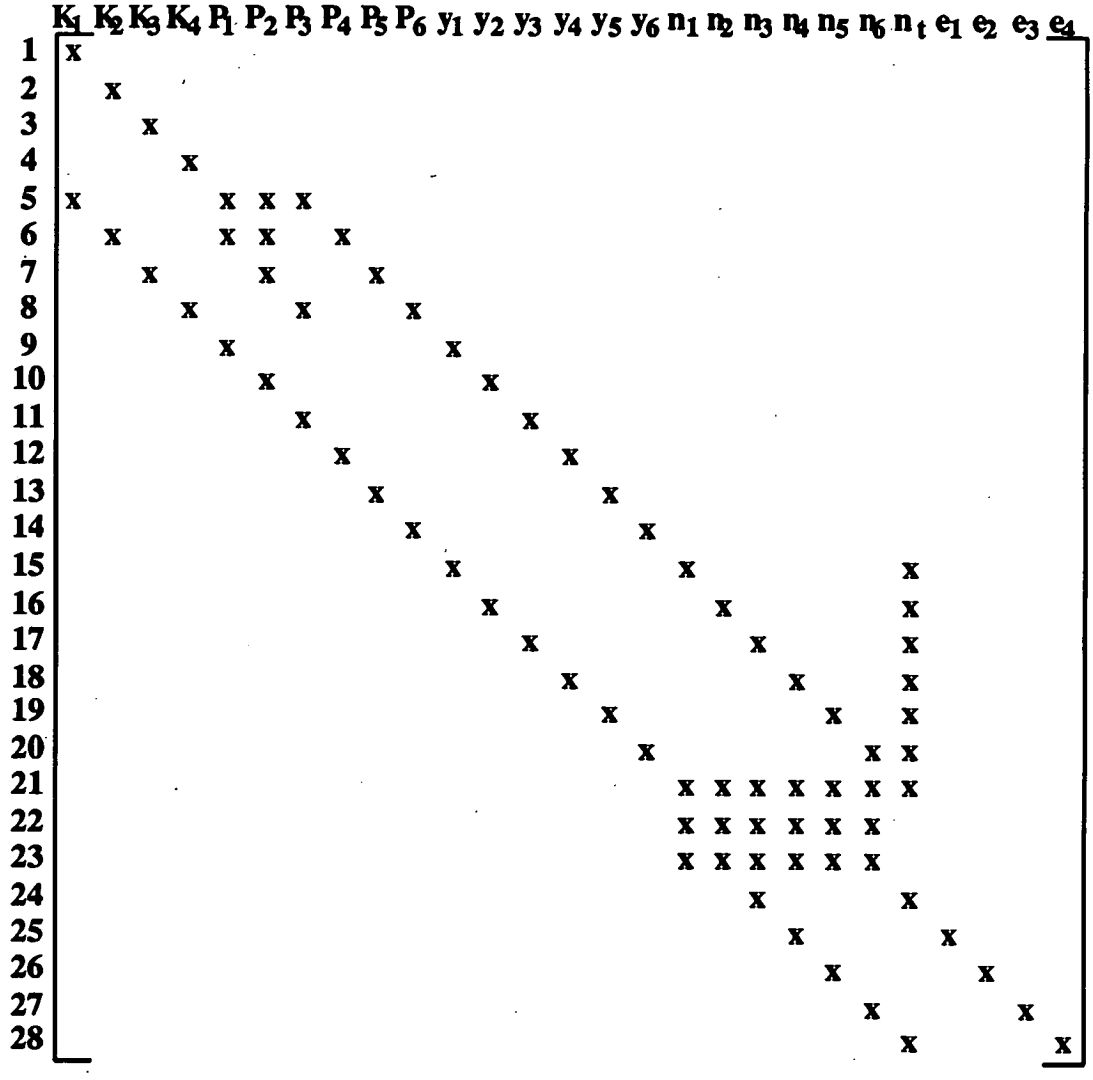

Figure C.3: The Incidence Matrix for the Dissociation Equations

Appendix D

Methods for Convergence Acceleration

D.1 Derivative Methods

D.1.1 Methods with an Analytical Jacobian

The general form for any method in which the correction to the solution vector at the i^{th} iteration, Δx^i, is a linear transformation of the residual vector function, f^i, is

$$\Delta x^i = -\alpha^i B^i f^i \tag{D.1}$$

where $0 \leq \alpha^i \leq 1$, $|\Delta x^i| = |f^i| = m$, and B^i is an $m \times m$ non-singular matrix. If the Jacobian of f^i is J^i, then

$$B^i = (J^i)^{-1} \tag{D.2}$$

corresponds to equation D.1 being the Newton-Raphson method; if instead

$$B^i = (J^i)^{-T} \tag{D.3}$$

is used, it is the method of steepest descent [42]. If B^i is taken to be a linear combination of $(J^i)^{-1}$ and $(J^i)^{-T}$,

$$B^i = (J^i)^{-T}((J^i)^{-T}(J^i)^{-1} + \lambda^i(J^i)^{-1}),\ 0 \leq \lambda^i \leq 1 \qquad \text{(D.4)}$$

then equation D.1 corresponds to a method attributed both to Levenberg [56] and Marquardt [62].

D.1.2 Methods which Use Function Values

If the Jacobian is either difficult or expensive to calculate analytically, then it may be approximated numerically by perturbing each component of the solution vector in turn. Then $B^i = \{b^i_{jk}\}$ becomes

$$b^i_{jk} = \frac{f^i_j(x^i + h_k e_k) - f^i_j(x^i)}{h_k},\ j,k = 1,2,\cdots,m \qquad \text{(D.5)}$$

where f^i_j is the j^{th} component of f^i, e_k is the k^{th} column of I_m and h_k is some number much smaller than one.

Wegstein [99] developed the secant method for solving single variable problems, and his method has been generalised to the simultaneous solution of m equations. In his method, initial guesses are required for x^0 and x^1 and then at each iteration

$$b^i_{jk} = \frac{f^i_j(x^{i+1}_k) - f^i_j(x^i_k)}{x^{i+1}_k - x^i_k} \qquad \text{(D.6)}$$

Here the variables are regarded as independent of one another and the functions f^i are approximated by the linear equations which intersect with them at x^i_k and x^{i+1}_k.

D.2 Quasi-Newton Methods

A different approach is for B^i in D.1 to be an approximation to the Jacobian which is improved after each iteration. B^0 can be chosen to be an arbitrary

matrix, but more usually it is selected as either I_m, the identity matrix for $\mathbb{R}^m$, or the Jacobian of $f(x^0)$. Whatever the choice of B^0, equation D.1 is generally not satisfied. Instead, the update to B^i is chosen so that

$$B^{i+1}\Delta x^i = \Delta f^i \tag{D.7}$$

where $\Delta f^i = f^{i+1} - f^i$. Since B^{i+1} is chosen to reflect the change in the value of f^i along the direction of Δx^i, the modification to B^i need only be of rank one. Thus the general form for the change in the iteration matrix is

$$B^{i+1} = B^i + uv^T \tag{D.8}$$

where u and v are column vectors. Substituting this equation into D.7 gives

$$\left\{B^i + uv^T\right\}\Delta x^i = \Delta f^i \tag{D.9}$$

from which it can be deduced that u must lie in the direction of $\Delta f^i - B^i\Delta x^i$, *i.e.*

$$u = \frac{\Delta f^i - B^i\Delta x^i}{v^T\Delta x^i} \tag{D.10}$$

The vector v^T has been chosen in different ways. Barnes [10], chose it to be orthogonal to each Δx^j, $j < i$, wheras Broyden [13], [14] chose it to be Δx^i in order to preserve the positive definiteness of the iteration matrix.

D.3 Dominant Eigenvalue Methods

Consider the linear equation set

$$Ax = b \tag{D.11}$$

where x is the vector of unknowns, A is the unsymmetric matrix of real coefficients and b is the vector of right hand sides. Equation D.11 can be solved exactly as $x = A^{-1}b$, or iteratively if A is modified in some way. Without loss of generality, let A be of dimension $m \times m$, and rewrite it as $A = B - C$ where $B = \{b_{ij}\}$ and $b_{ij} = a_{ij}$, $i = 1, 2\cdots, m$, $j < i$. Then D.11 may be transformed into the iterative scheme

$$x^{k+1} = \Gamma x^k + \nu \tag{D.12}$$

where $\Gamma = B^{-1}C$ and $\nu = B^{-1}b$; *n.b.* Γ is independent of x^k. If $x^\star$ is the solution of D.11, then

$$x^\star = \Gamma x^\star + \nu \tag{D.13}$$

Rearranging this for ν and substituting the result into D.12 gives

$$x^{k+1} - x^\star = \Gamma(x^k - x^\star) \tag{D.14}$$

Clearly, Γ is a linear operator which relates the error in x^{k+1} to that in x^k. Writing equation D.14 for $k = 0$ gives

$$x^1 - x^* = \Gamma(x^0 - x^*) \tag{D.15}$$

and so, following the next iteration,

$$x^2 - x^* = \Gamma^2(x^0 - x^*) \tag{D.16}$$

Repeated application of Γ shows that after the n^{th} iteration,

$$x^n - x^* = \Gamma^n(x^0 - x^*) \tag{D.17}$$

or, setting $e^i = x^i - x^*$,

$$e^n = \Gamma^n e^0 \tag{D.18}$$

If $\lambda_1, \lambda_2, \cdots, \lambda_m$, the eigenvalues of Γ, are real and distinct, then $\hat{z}_i \in \hat{\Phi}$, the set of the eigenvectors of Γ, is orthogonal to each other member of this set. Further, each of these vectors can be scaled so that $z_i = \frac{\hat{z}_i}{\hat{z}_i^T \hat{z}_i}, i = 1, 2, \cdots, m$ is an orthonormal basis for $\mathbb{R}^m$, and the vector e^0 can be written as a linear combination of these scaled vectors. Hence

$$e^0 = \sum_{i=1}^{i=m} \alpha_i z_i \tag{D.19}$$

Premultiplying D.19 by $z_j, 1 \leq j \leq m$, and using the orthonormality of the eigenvectors gives

$$\alpha_i = z_i^T e^0 \tag{D.20}$$

Substituting this expression into D.19, and the result into D.18 gives

$$e^n = \Gamma^n \sum_{i=1}^{i=m} z_i^T e^0 z_i \tag{D.21}$$

However, for each eigenvector z_l of Γ and the coresponding eigenvalue λ_l,

$$\Gamma z_l = \lambda_l z_l \tag{D.22}$$

and so D.21 becomes

$$e^n = \sum_{i=1}^{i=m} z_i^T e^0 \lambda_i^n z_i \tag{D.23}$$

If the eigenvalues of Γ are ranked in order of decreasing absolute value, *i.e.* $|\lambda_1| > |\lambda_2| > \cdots > |\lambda_m|$, then the error vector after the k^{th} iterate is

$$e^k = \lambda_1^k \{z_1^T e^0 z_1 + \sum_{i=2}^{i=m} z_i^T e^0 (\frac{\lambda_i}{\lambda_1})^k z_i\} \tag{D.24}$$

A sufficient but not necessary condition for this iterative scheme to converge is that the modulus of λ_1 should be less than unity. Whether this holds or not, if $|\lambda_1| \gg |\lambda_2|$, then the first term in this equation will tend to dominate. In this case the approximation

$$\lim_{k\to\infty} e^k = \lambda_1^k z_1^T e^0 z_1 \tag{D.25}$$

holds and the difference between this approximation and the true value of e^k follows a decreasing geometric series. λ_1 is then said to be the dominant eigenvalue of Γ.

The derivation of an approximation for the error in the solution at the k^{th} iteration depended on the symmetry of A. Should A be unsymmetric, however, then the detail of this derivation changes, but not so the essence. In this case the left and right hand eigenvectors of Γ should each form an orthonormal basis for $\mathbb{R}^m$. This requires that

$$w_j^T z_i = \begin{cases} 1 & i = j \\ 0 & i \neq j \end{cases} \tag{D.26}$$

where w_j is the j^{th} left hand eigenvector of Γ, and the left and right hand eigenvectors are ordered in the same way. Further, if A is unsymmetric then w_i^T replaces z_i^T in equation D.21. Should the first absolute values of the first β eigenvalues of Γ be similar, then the first β terms of the summation must be included in equation D.25. If Γ has one or more multiple eigenvalues, then the above treatment still holds if there exists some non-singular matrix H such that

$$H^{-1}\Gamma H = diag(\lambda_i) \tag{D.27}$$

where $diag(\lambda_i)$ is a diagonal matrix whose non-zero entries are the eigenvalues of Γ. In this case the eigenvectors which correspond to equivalent eigenvalues are non-unique, but an orthonormal subset of them which spans $\mathbb{R}^m$ can still be chosen. If the dominant eigenvalue is not unique, then the convergence of D.25 is reduced. Further, if some of the eigenvalues are complex conjugate pairs[1] then the corresponding terms in D.23 oscillate and this impairs the rate of convergence of D.25.

The above analysis can be extended to the stationary, iterative solution of non-linear equations. In this case, equation D.18 becomes

$$e^k = \Gamma^k e^{k-1} \tag{D.28}$$

where Γ^k may change at each iteration. However, if the ratio of successive errors begins to follow a geometric progression, it may be that the functions over the

---

[1] *n.b.* since A is real so too is Γ, and thus because $det(\Gamma) = \prod_{i=1}^{i=m} \lambda_i$, for any complex eigenvalue of Γ, λ_{j1}, there must exist also $\lambda_{j2} = \bar{\lambda_{j1}}$

domain defined by these iterations is sufficiently linear for Γ^k to be regarded as independent of k. Then the acceleration steps described in § D.4 may be used to promote convergence.

D.4 Application to Convergence Acceleration

Aitken [4] noted that since both the error in x and its change at each iteration follow a geometric series whose common ratio is λ_1, then his acceleration method for the calculation of eigenvalues [3] could be used to accelerate the convergence of linear equations. Let the change in x over the k^{th} iteration be

$$\Delta x^k = x^{k+1} - x^k \tag{D.29}$$

Then Aitken's observation may be written as

$$\frac{e^k}{e^{k-1}} \approx \frac{\Delta x^k}{\Delta x^{k-1}} \tag{D.30}$$

If equations D.18 and D.29 are substituted into D.30, the result may be rearranged to give an as approximation to the $x^\star$, the solution of the equations,

$$x^\star \approx \frac{x^{k-1}x^{k+1} - (x^k)^2}{x^{k-1} - 2x^k - x^{k+1}} \tag{D.31}$$

Rather than waiting until the approximation D.30 is sufficiently small, the author advocates testing the change in the value of each component of x until this approaches a geometric series and then taking an acceleration step

$$x_i^\star \approx \frac{x_i^{k-1}x_i^{k+1} - (x_i^k)^2}{x_i^{k-1} - 2x_i^k - x_i^{k+1}} \tag{D.32}$$

where x_i^k is the i^{th} component of x^k. This accelerates each component of x by a different amount and, because it ignores the effect of interaction between variables, it can lead to oscillation.

Orbach and Crowe [67] estimate the modulus of the dominant eigenvalue as the ratio of absolute value of the change in x over successive iterations, and determine its sign by comparing the elements of x between them. Their application of dominant eigenvalue methods is for the solution of nonlinear equations and the convergence of flowsheets and they assume that D.25 holds when the condition

$$1 - \epsilon \leq \frac{|\Delta x_l^{k+1}|}{|\Delta x_l^k|} \leq 1 + \epsilon \tag{D.33}$$

is satisfied, where Δx_i^k is the maximum change of any component of x over the k^{th} iteration, and ϵ is some small number. Assuming that k is close enough to the limit in equation D.25 for the equality to hold, these authors extrapolated the solution to the equations to

$$x^\star = x^k + \alpha \frac{x^{k+1} - x^k}{1 - \lambda_1} \tag{D.34}$$

where $0 < \alpha \leq$, and this variable is included in order to supress oscillation of the solution.

This method is called the Dominant Eigenvalue Method, and it is least effective when there are more than one eigenvalues close to unity, and which dominate the rate of convergence. Crowe and Nishio [18] sought to alleviate this problem by taking account of the ν greatest eigenvalues of the iteration matrix A, where ν may be estimated in different ways. Their argument is based on the use of the Caley-Hamilton theorem [104] which states that a matrix behaves its own characteristic equation. They order the eigenvalues of A in descending order, and they use the relationship

$$\Delta x^k = \sum_{j=1}^{j=m} z_j \lambda_j^k \tag{D.35}$$

to form the approximation

$$\sum_{j=1}^{j=\nu} \hat{\mu}_j \Delta x_{i-j} \approx 0, \quad i = k, k+1, \cdots, \infty \tag{D.36}$$

where $\hat{\mu}_j$ is an approximation to the j^{th} coefficient in the characteristic equation of A, and $\hat{\mu}_0 = 0$. The coefficients in D.36 can be made to approximate the values of the true eigencoefficients by minimising the Euclidian norm of the left hand side of this equation. The current estimate of the solution, x^k, can be extrapolated to a new estimate, $\hat{x}^\infty$, by rearranging D.36 to give

$$\hat{x}^\infty = x^k - \frac{\sum_{i=0}^{i=\nu-1} \left(\sum_{j=i+1}^{j=\nu} \hat{\mu}_j \right) \Delta x^{k-i-1}}{\sum_{j=0}^{j=\nu} \hat{\mu}_j} \tag{D.37}$$

Crowe and Nishio call this the General Dominant Eigenvalue Method and they show that it reduces to the same form as the Dominant Eigenvalue Method [67] for $\nu = 1$. The authors [18] report that using the same criteria as Orbach and Crowe for determining when to take a promoting step resulted in too infrequent

acceleration. Instead they recommend that a promotion step be taken when the sum of the differences between the components of $\hat{x}^{\infty}$ estimated at successive iterations is less than some small value ϵ. Whilst this criterion is redolent of Aitken's [4], Crowe and Nishio's acceleration procedure is superior to his in that their's takes account of both the interaction between variables and a larger subset of the dominating eigenvalues in the iteration matrix.

Appendix E

The Modelling Interpreter

```
/*

   Program : prob_interp

   Author  : A. T. Doig

   Date    : 19th February 1990

   Purpose : In order to minimise the problems of memory exhaustion and
             excessive search times the model producing software is
             written so that each predicate fails rather than succeed. In
```

order to add flexibility to our approach a general interpreter is provided which can be used to control any set of predicates. This interpreter is the predicate program/0 and it consists of three separate rules :

[1] Assert to the database the name of the file which contains the program description, that is the list of tasks to be performed and the files which contain the predicates necessary to complete them. At the same time assert the list of predicates which are necessary only for setting the system up - these will be retracted from the database immediately after use.

[2] Call each task in turn and on completion of this task remove all of the predicates which are no longer of any use.

[3] Leave Prolog.

Obviously this program is not the most concise description of the problem which is possible using predicate calculus. It has the advantage, however, of combining a relatively high degree of conciseness with both simplicity of implementation and clarity.

*/

/*

```
   program/0 is the interpreter. Assert the information about set up to the
   database and then fail so as to free the heap. Next call each task in turn
   and then leave Prolog.

*/

program:-
        consult(prob_descrip),
        initialise_system.
program:-
        call_tasks.
program:-
        halt.

/*

   call_tasks/0 uses the predicate repeat/0 to force backtracking each time
   that next_task/0 fails. The last time that next_task/0 is called it
   succeeds (there are no more tasks to be fulfilled), and the cut-fail
   combination is used to disable repeat/0 and cause call_tasks/0 to fail.

*/

call_tasks:-
        repeat,
        next_task,
        !, fail.

/*
```

```
   next_task/0 retracts from the database the name of the next task (the
   predicate to be called), and the names of the files which contain the
   code necessary for its completion. When the task has been completed
   complete_task/2 fails and so the next call to next_task/0 is made. This
   call identifies the set of predicates which are no longer of any use
   and these are retracted from the database by house_keep/1. This last
   predicate also fails on completion and so, because of the cut, next_task/0
   fails completely, returning control to call_tasks/0 on backtracking.

   Eventually next_task/0 is called when no structures of the form
   task_list/2 or pred_set/1 remain in the database. In this case next_task/0
   succeeds.

*/

next_task:-
        not(recorded(preds, task_list(_, _), _)), !.
next_task:-
        my_retract(preds, task_list(Files, Task)),
        complete_task(Files, Task).
next_task:-
        house_keep.

/*

   complete_task/2 uses its arguements in two seperate clauses. First of all
   the list of files which is its first arguement is consulted and then
   the task defined by its second arguement is called. Both consult_all/1
```

```
   and the task called fail.

*/

complete_task(Files, _):-
        consult_all(Files).
complete_task(_, Task):-
        call(Task).

/*

   consult_all/1 reconsults the file at the head of the list which is its
   first arguement and then makes a call to reconsult all of those in the
   tail.

*/

consult_all([H|T]):-
        !, reconsult(H),
        consult_all(T).
consult_all([]):-fail.

/*

   house_keep/0 removes all of the redundant clause from the database. It
   does this by calling abolish/2 for each of these clauses in turn. The
   first arguement is the functor of the clause to removed and the second
   is its arity.
```

```
*/

house_keep:-
        repeat,
        not(kill_pred),
        !, fail.

kill_pred:-
        retract(dead_pred(P, A)),
        abolish(P, A).
```

Appendix F

The Initialization File for the Flash Problem

This is a copy of the file *pol_init.c* which is written by the modelling software. As was explained in § 7.5.1 this file contains the logic necessary for declaring the subroutines of *pol_eval.c* for main, for initialising pointers to those subroutines and for storing the values of the constants and guesses for the tear variables. It also contains a switch which is used to access the tear variables for each subroutine.

```
char *malloc();
char *free();

initialise_prob(vals, Iters, N_probs, f_ptr, val_size)
```

```
double **vals;      /* The unknowns and constants in the problem.          */
int **Iters;        /* The array of iterations required for each partition. */
int *N_probs;       /* The number of partitions in the problem.            */
int (*f_ptr[])();   /* The array of pointers to the evaluation functions   */
int *val_size;      /* The number of variables/knowns                      */
-

  int part_num = 18;
  int prob_size = 68;

  int eval_1();
  int eval_2();
  int eval_3();
  int eval_4();
  int eval_5();
  int eval_6();
  int eval_7();
  int eval_8();
  int eval_9();
  int eval_10();
  int eval_11();
  int eval_12();
  int eval_13();
  int eval_14();
  int eval_15();
  int eval_16();
  int eval_17();
  int eval_18();

  f_ptr[1] = eval_1;
  f_ptr[2] = eval_2;
  f_ptr[3] = eval_3;
  f_ptr[4] = eval_4;
  f_ptr[5] = eval_5;
  f_ptr[6] = eval_6;
  f_ptr[7] = eval_7;
  f_ptr[8] = eval_8;
  f_ptr[9] = eval_9;
  f_ptr[10] = eval_10;
  f_ptr[11] = eval_11;
  f_ptr[12] = eval_12;
  f_ptr[13] = eval_13;
  f_ptr[14] = eval_14;
  f_ptr[15] = eval_15;
  f_ptr[16] = eval_16;
  f_ptr[17] = eval_17;
  f_ptr[18] = eval_18;

  *val_size = 68;

/*
  Set up the arrays and initialise them. Recall that the +1 bit is necessary
  because C arrays start at subscript 0, rather than 1.

*/
```

```
*vals = (double *) malloc((unsigned) (*val_size+1) * sizeof(double));
*Iters = (int *) malloc((unsigned) (part_num+1) * sizeof(int));

**vals = prob_size;
**Iters = part_num;

*N_probs = part_num;

  *((*vals) + 1) = 348.5;
  *((*vals) + 2) = 0.422;
  *((*vals) + 3) = 0.4;
  *((*vals) + 4) = 0,3;
  *((*vals) + 5) = 100.0;
  *((*vals) + 6) = 40683.0;
  *((*vals) + 7) = 38770.0;
  *((*vals) + 8) = 35278.0;
  *((*vals) + 9) = -242000.0;
  *((*vals) + 10) = -234960.0;
  *((*vals) + 11) = -201300.0;
  *((*vals) + 12) = 75.3;
  *((*vals) + 13) = 97.1;
  *((*vals) + 14) = 80.4;
  *((*vals) + 15) = 298.0;
  *((*vals) + 16) = 298.0;
  *((*vals) + 17) = 1.0;
  *((*vals) + 18) = 0.81564;
  *((*vals) + 19) = 0.94934;
  *((*vals) + 20) = 0.20022;
  *((*vals) + 21) = 1.0;
  *((*vals) + 22) = 0.60908;
  *((*vals) + 23) = 0.43045;
  *((*vals) + 24) = 1.35386;
  *((*vals) + 25) = 1.0;
  *((*vals) + 26) = -46.13;
  *((*vals) + 27) = -41.68;
  *((*vals) + 28) = -34.29;
  *((*vals) + 29) = 3816.44;
  *((*vals) + 30) = 3803.98;
  *((*vals) + 31) = 3626.55;
  *((*vals) + 32) = 18.3036;
  *((*vals) + 33) = 18.9119;
  *((*vals) + 34) = 18.5875;
  *((*vals) + 40) = 0.25;
  *((*vals) + 47) = 60.0;
  *((*vals) + 41) = 0.6;

/*

  get_eval/3 uses a switch to assign the function pointer f_ptr correctly
  and to set up tear_list.

*/
```

```
get_eval(Prob, tear_list)

int Prob;          /* The current partition                    */
int **tear_list;   /* The list of tear variables.              */

{

/* Flip the switch */

  switch(Prob) {

    case 1:

      *tear_list = (int *) malloc((unsigned) 1 * sizeof(int));
      *((*tear_list)) = 0;

    break;

    case 2:

      *tear_list = (int *) malloc((unsigned) 1 * sizeof(int));
      *((*tear_list)) = 0;

    break;

    case 3:

      *tear_list = (int *) malloc((unsigned) 1 * sizeof(int));
      *((*tear_list)) = 0;

    break;

    case 4:

      *tear_list = (int *) malloc((unsigned) 1 * sizeof(int));
      *((*tear_list)) = 0;

    break;

    case 5:

      *tear_list = (int *) malloc((unsigned) 1 * sizeof(int));
      *((*tear_list)) = 0;

    break;

    case 6:

      *tear_list = (int *) malloc((unsigned) 1 * sizeof(int));
      *((*tear_list)) = 0;

    break;

    case 7:

      *tear_list = (int *) malloc((unsigned) 1 * sizeof(int));
```

```
    *((*tear_list)) = 0;

  break;

  case 8:

    *tear_list = (int *) malloc((unsigned) 1 * sizeof(int));
    *((*tear_list)) = 0;

  break;

  case 9:

    *tear_list = (int *) malloc((unsigned) 1 * sizeof(int));
    *((*tear_list)) = 0;

  break;

  case 10:

    *tear_list = (int *) malloc((unsigned) 1 * sizeof(int));
    *((*tear_list)) = 0;

  break;

  case 11:

    *tear_list = (int *) malloc((unsigned) 1 * sizeof(int));
    *((*tear_list)) = 0;

  break;

  case 12:

    *tear_list = (int *) malloc((unsigned) 1 * sizeof(int));
    *((*tear_list)) = 0;

  break;

  case 13:

    *tear_list = (int *) malloc((unsigned) 1 * sizeof(int));
    *((*tear_list)) = 0;

  break;

  case 14:

    *tear_list = (int *) malloc((unsigned) 1 * sizeof(int));
    *((*tear_list)) = 0;

  break;

  case 15:

    *tear_list = (int *) malloc((unsigned) 4 * sizeof(int));
```

```
  *((*tear_list)) = 3;
  *((*tear_list) + 1) = 40;
  *((*tear_list) + 2) = 47;
  *((*tear_list) + 3) = 41;

break;

case 16:

  *tear_list = (int *) malloc((unsigned) 1 * sizeof(int));
  *((*tear_list)) = 0;

break;

case 17:

  *tear_list = (int *) malloc((unsigned) 1 * sizeof(int));
  *((*tear_list)) = 0;

break;

case 18:

  *tear_list = (int *) malloc((unsigned) 1 * sizeof(int));
  *((*tear_list)) = 0;

break;

default :

  printf("%s"n"n", "No partitions - computation abandoned");
  exit(-1);

break;

"
```

Appendix G

The Subroutines for the Flash Problem

```
#include "consts.h"
#include "arith.h"

int push();
double pop();
```

```
char *malloc();
char *free();

int eval_1(vals)
double vals[];
{
vals[64]=(vals[16]-vals[15])*vals[12]+vals[9];
}

int eval_2(vals)
double vals[];
{
vals[63]=(vals[16]-vals[15])*vals[13]+vals[10];
}

int eval_3(vals)
double vals[];
{
vals[62]=(vals[16]-vals[15])*vals[14]+vals[11];
}

int eval_4(vals)
double vals[];
{
vals[60]=(vals[1]-vals[15])*vals[12]+vals[9];
}

int eval_5(vals)
double vals[];
{
vals[59]=(vals[1]-vals[15])*vals[13]+vals[10];
}

int eval_6(vals)
double vals[];
{
vals[58]=(vals[1]-vals[15])*vals[14]+vals[11];
}

int eval_7(vals)
double vals[];
{
vals[56]=vals[60]+vals[6];
}

int eval_8(vals)
double vals[];
{
vals[55]=vals[59]+vals[7];
}

int eval_9(vals)
double vals[];
{
vals[54]=vals[58]+vals[8];
}
```

```
int eval_10(vals)
double vals[];
{
vals[52]=exp(vals[32]-vals[29]/(vals[1]+vals[26]));
}

int eval_11(vals)
double vals[];
{
vals[51]=exp(vals[33]-vals[30]/(vals[1]+vals[27]));
}

int eval_12(vals)
double vals[];
{
vals[50]=exp(vals[34]-vals[31]/(vals[1]+vals[28]));
}

int eval_13(vals)
double vals[];
{
vals[35]=1.0-vals[4]-vals[3];
}

int eval_14(vals)
double vals[];
{
vals[61]=vals[62]*vals[3]+vals[63]*vals[35]+vals[64]*vals[4];
}

int eval_15(vals, f, jacobian)

double vals[];       /* The unknowns and constants in the problem. */
double f[];          /* The array of function values.              */
double **jacobian;   /* The jacobian for the tear set.             */

{

  double *stack;              /* The stack                          */
  double **s_dash;            /* The derivative stacks              */
  double **chain;             /* The analytical derivatives         */
  double **unit;              /* The identity matrix (Dx/Dx)        */
  double pop();               /* The popping function               */
  double *s_row;              /* An array of zeros                  */


  int *s_ptr;                 /* A pointer to the head of the stack */
  int C;                      /* The number of tear variables       */
  int dep;                    /* The number of dependent variables  */
  int exit_flag = 1;          /* Flag unused at the moment          */
  int i, j;                   /* Count variables                    */

dep = 14;
```

```
C = (int) **jacobian;

stack  = (double *) malloc((unsigned) (MAX_PTR+1) * sizeof(double));

s_dash = (double **) malloc((unsigned) (MAX_PTR+1) * sizeof(double *));

chain  = (double **) malloc((unsigned) (dep+1) * sizeof(double *));

unit   = (double **) malloc((unsigned) (C+1) * sizeof(double *));

s_row  = (double *) malloc((unsigned) (C+1) * sizeof(double));

for(i=1; i <= C; i++) -

  s_row[i] = 0.0;

  unit[i]  = (double *) malloc((unsigned) (C+1) * sizeof(double));

  for(j=1; j <= C; j++)
    *(*(unit + i) + j) = 0.0;

  *(*(unit + i) + i) = 1.0;

-


for(i=1; i <= MAX_PTR; i++) -

 s_dash[i] = (double *) malloc((unsigned) (C+1) * sizeof(double));

-

for(i=1; i <= dep; i++) -

  chain[i]  = (double *) malloc((unsigned) (C+1) * sizeof(double));

    for(j=1; j <= C; j++)
      *(*(chain + i) + j) = 0.0;

-


*s_dash = (double *) malloc((unsigned) sizeof(double));

*chain = (double *) malloc((unsigned) sizeof(double));

*unit = (double *) malloc((unsigned) sizeof(double));

stack[0] = vals[0];

s_row[0] = **jacobian;

s_ptr = (int *) malloc((unsigned) sizeof(int));

**s_dash = f[0];
```

```
  (*s_ptr) = 0;

push(vals[41], unit[3], stack, s_dash, s_ptr);
push(vals[23], s_row, stack, s_dash, s_ptr);
push(vals[2], s_row, stack, s_dash, s_ptr);
my_times(stack, s_dash, s_ptr);
minus(stack, s_dash, s_ptr);
push(vals[24], s_row, stack, s_dash, s_ptr);
push(vals[40], unit[1], stack, s_dash, s_ptr);
my_times(stack, s_dash, s_ptr);
minus(stack, s_dash, s_ptr);
push(vals[25], s_row, stack, s_dash, s_ptr);
divide(stack, s_dash, s_ptr);

vals[39] = pop(1, stack, s_dash, chain, s_ptr);

push(vals[5], s_row, stack, s_dash, s_ptr);
push(vals[47], unit[2], stack, s_dash, s_ptr);
minus(stack, s_dash, s_ptr);

vals[68] = pop(2, stack, s_dash, chain, s_ptr);

push(vals[5], s_row, stack, s_dash, s_ptr);
push(vals[4], s_row, stack, s_dash, s_ptr);
my_times(stack, s_dash, s_ptr);
push(vals[68], chain[2], stack, s_dash, s_ptr);
push(vals[2], s_row, stack, s_dash, s_ptr);
my_times(stack, s_dash, s_ptr);
minus(stack, s_dash, s_ptr);
push(vals[47], unit[2], stack, s_dash, s_ptr);
divide(stack, s_dash, s_ptr);

vals[38] = pop(3, stack, s_dash, chain, s_ptr);

push(vals[22], s_row, stack, s_dash, s_ptr);
push(vals[39], chain[1], stack, s_dash, s_ptr);
my_times(stack, s_dash, s_ptr);
push(vals[21], s_row, stack, s_dash, s_ptr);
push(vals[40], unit[1], stack, s_dash, s_ptr);
my_times(stack, s_dash, s_ptr);
plus(stack, s_dash, s_ptr);
push(vals[20], s_row, stack, s_dash, s_ptr);
push(vals[2], s_row, stack, s_dash, s_ptr);
my_times(stack, s_dash, s_ptr);
plus(stack, s_dash, s_ptr);

vals[42] = pop(4, stack, s_dash, chain, s_ptr);

push(vals[19], s_row, stack, s_dash, s_ptr);
push(vals[39], chain[1], stack, s_dash, s_ptr);
my_times(stack, s_dash, s_ptr);
push(vals[18], s_row, stack, s_dash, s_ptr);
push(vals[40], unit[1], stack, s_dash, s_ptr);
my_times(stack, s_dash, s_ptr);
plus(stack, s_dash, s_ptr);
push(vals[17], s_row, stack, s_dash, s_ptr);
```

```
push(vals[2], s'row, stack, s'dash, s'ptr);
my'times(stack, s'dash, s'ptr);
plus(stack, s'dash, s'ptr);

vals[43] = pop(5, stack, s'dash, chain, s'ptr);

push(1.0, s'row, stack, s'dash, s'ptr);
push(vals[17], s'row, stack, s'dash, s'ptr);
push(vals[2], s'row, stack, s'dash, s'ptr);
my'times(stack, s'dash, s'ptr);
push(vals[43], chain[5], stack, s'dash, s'ptr);
divide(stack, s'dash, s'ptr);
push(vals[20], s'row, stack, s'dash, s'ptr);
push(vals[40], unit[1], stack, s'dash, s'ptr);
my'times(stack, s'dash, s'ptr);
push(vals[42], chain[4], stack, s'dash, s'ptr);
divide(stack, s'dash, s'ptr);
plus(stack, s'dash, s'ptr);
push(vals[23], s'row, stack, s'dash, s'ptr);
push(vals[39], chain[1], stack, s'dash, s'ptr);
my'times(stack, s'dash, s'ptr);
push(vals[41], unit[3], stack, s'dash, s'ptr);
divide(stack, s'dash, s'ptr);
plus(stack, s'dash, s'ptr);
minus(stack, s'dash, s'ptr);

vals[46] = pop(6, stack, s'dash, chain, s'ptr);

push(vals[46], chain[6], stack, s'dash, s'ptr);
expon(stack, s'dash, s'ptr);
push(vals[43], chain[5], stack, s'dash, s'ptr);
divide(stack, s'dash, s'ptr);

vals[67] = pop(7, stack, s'dash, chain, s'ptr);

push(vals[67], chain[7], stack, s'dash, s'ptr);
push(vals[2], s'row, stack, s'dash, s'ptr);
my'times(stack, s'dash, s'ptr);
push(vals[52], s'row, stack, s'dash, s'ptr);
my'times(stack, s'dash, s'ptr);
push(vals[38], chain[3], stack, s'dash, s'ptr);
divide(stack, s'dash, s'ptr);

vals[49] = pop(8, stack, s'dash, chain, s'ptr);

push(1.0, s'row, stack, s'dash, s'ptr);
push(vals[18], s'row, stack, s'dash, s'ptr);
push(vals[2], s'row, stack, s'dash, s'ptr);
my'times(stack, s'dash, s'ptr);
push(vals[43], chain[5], stack, s'dash, s'ptr);
divide(stack, s'dash, s'ptr);
push(vals[21], s'row, stack, s'dash, s'ptr);
push(vals[40], unit[1], stack, s'dash, s'ptr);
my'times(stack, s'dash, s'ptr);
push(vals[42], chain[4], stack, s'dash, s'ptr);
divide(stack, s'dash, s'ptr);
```

```
plus(stack, s'dash, s'ptr);
push(vals[24], s'row, stack, s'dash, s'ptr);
push(vals[39], chain[1], stack, s'dash, s'ptr);
my'times(stack, s'dash, s'ptr);
push(vals[41], unit[3], stack, s'dash, s'ptr);
divide(stack, s'dash, s'ptr);
plus(stack, s'dash, s'ptr);
minus(stack, s'dash, s'ptr);

vals[45] = pop(9, stack, s'dash, chain, s'ptr);

push(vals[45], chain[9], stack, s'dash, s'ptr);
expon(stack, s'dash, s'ptr);
push(vals[42], chain[4], stack, s'dash, s'ptr);
divide(stack, s'dash, s'ptr);

vals[66] = pop(10, stack, s'dash, chain, s'ptr);

push(vals[66], chain[10], stack, s'dash, s'ptr);
push(vals[40], unit[1], stack, s'dash, s'ptr);
my'times(stack, s'dash, s'ptr);
push(vals[51], s'row, stack, s'dash, s'ptr);
my'times(stack, s'dash, s'ptr);
push(vals[49], chain[8], stack, s'dash, s'ptr);
divide(stack, s'dash, s'ptr);

vals[37] = pop(11, stack, s'dash, chain, s'ptr);

push(1.0, s'row, stack, s'dash, s'ptr);
push(vals[38], chain[3], stack, s'dash, s'ptr);
minus(stack, s'dash, s'ptr);
push(vals[37], chain[11], stack, s'dash, s'ptr);
minus(stack, s'dash, s'ptr);

vals[36] = pop(12, stack, s'dash, chain, s'ptr);

push(1.0, s'row, stack, s'dash, s'ptr);
push(vals[19], s'row, stack, s'dash, s'ptr);
push(vals[2], s'row, stack, s'dash, s'ptr);
my'times(stack, s'dash, s'ptr);
push(vals[43], chain[5], stack, s'dash, s'ptr);
divide(stack, s'dash, s'ptr);
push(vals[22], s'row, stack, s'dash, s'ptr);
push(vals[40], unit[1], stack, s'dash, s'ptr);
my'times(stack, s'dash, s'ptr);
push(vals[42], chain[4], stack, s'dash, s'ptr);
divide(stack, s'dash, s'ptr);
plus(stack, s'dash, s'ptr);
push(vals[25], s'row, stack, s'dash, s'ptr);
push(vals[39], chain[1], stack, s'dash, s'ptr);
my'times(stack, s'dash, s'ptr);
push(vals[41], unit[3], stack, s'dash, s'ptr);
divide(stack, s'dash, s'ptr);
plus(stack, s'dash, s'ptr);
minus(stack, s'dash, s'ptr);
```

```
vals[44] = pop(13, stack, s_dash, chain, s_ptr);

push(vals[49], chain[8], stack, s_dash, s_ptr);
push(vals[36], chain[12], stack, s_dash, s_ptr);
my_times(stack, s_dash, s_ptr);
push(vals[50], s_row, stack, s_dash, s_ptr);
divide(stack, s_dash, s_ptr);
push(vals[39], chain[1], stack, s_dash, s_ptr);
divide(stack, s_dash, s_ptr);

vals[65] = pop(14, stack, s_dash, chain, s_ptr);

push(vals[5], s_row, stack, s_dash, s_ptr);
push(vals[35], s_row, stack, s_dash, s_ptr);
my_times(stack, s_dash, s_ptr);
push(vals[47], unit[2], stack, s_dash, s_ptr);
push(vals[37], chain[11], stack, s_dash, s_ptr);
my_times(stack, s_dash, s_ptr);
minus(stack, s_dash, s_ptr);
push(vals[68], chain[2], stack, s_dash, s_ptr);
divide(stack, s_dash, s_ptr);
push(vals[40], unit[1], stack, s_dash, s_ptr);
minus(stack, s_dash, s_ptr);

f[1] = pop(1, stack, s_dash, jacobian, s_ptr);

push(vals[5], s_row, stack, s_dash, s_ptr);
push(vals[3], s_row, stack, s_dash, s_ptr);
my_times(stack, s_dash, s_ptr);
push(vals[68], chain[2], stack, s_dash, s_ptr);
push(vals[39], chain[1], stack, s_dash, s_ptr);
my_times(stack, s_dash, s_ptr);
minus(stack, s_dash, s_ptr);
push(vals[36], chain[12], stack, s_dash, s_ptr);
divide(stack, s_dash, s_ptr);
push(vals[47], unit[2], stack, s_dash, s_ptr);
minus(stack, s_dash, s_ptr);

f[2] = pop(2, stack, s_dash, jacobian, s_ptr);

push(vals[44], chain[13], stack, s_dash, s_ptr);
expon(stack, s_dash, s_ptr);
push(vals[65], chain[14], stack, s_dash, s_ptr);
divide(stack, s_dash, s_ptr);
push(vals[41], unit[3], stack, s_dash, s_ptr);
minus(stack, s_dash, s_ptr);

f[3] = pop(3, stack, s_dash, jacobian, s_ptr);


  for(i=0; i <= MAX_PTR; i++)
    free((char *) s_dash[i]);

  for(i=0; i <= dep; i++)
    free((char *) chain[i]);
```

```
  for(i=0; i <= C; i++)
    free((char *) unit[i]);

  free((char *) stack);

  free((char *) chain);

  free((char *) s_dash);

  free((char *) unit);

  return(exit_flag);

-
int eval_16(vals)
double vals[];
-
vals[53]=vals[54]*vals[36]+vals[55]*vals[37]+vals[56]*vals[38];
-

int eval_17(vals)
double vals[];
-
vals[57]=vals[58]*vals[39]+vals[59]*vals[40]+vals[60]*vals[2];
-

int eval_18(vals)
double vals[];
-
vals[48]=vals[5]*vals[61]-(vals[68]*vals[57]+vals[47]*vals[53]);
-
```